СОВРЕМЕННАЯ АРХИТЕКТУРА И УСТРОЙСТВО КОМПЬЮТЕРОВ

Вы разработчик программного обеспечения, системный архитектор или студент, изучающий устройство компьютеров? Вы ищете издание, которое подробно познакомит вас с внутренней архитектурой и принципами работы цифровых устройств? В этом пошаговом руководстве на практических примерах вы узнаете, как работают современные компьютерные системы, виртуальные машины, смартфоны и облачные серверы. Вы получите представление о внутреннем устройстве процессоров и поймете, как они выполняют код, написанный на языках высокого уровня.

Из этой книги вы узнаете об основах современных компьютерных систем, подробно изучите устройство процессоров, включая логические вентили, триггеры, регистры, последовательную логику, обработку прерываний и конвейеры управления. Вы сможете исследовать процессорные архитектуры и наборы инструкций, включая x86, x64, ARM и RISC-V. Вы узнаете, как реализовать процессор RISC-V на недорогой плате FPGA (ПЛИС), а также напишете программу для квантовых вычислений и запустите ее на реальном квантовом компьютере.

Второе издание включает в себя описание архитектуры и принципов проектирования, лежащих в основе таких важных областей, как кибербезопасность, блокчейн и майнинг криптовалют, а также управление беспилотными автомобилями. Прочитав эту книгу, вы получите глубокое представление о современных процессорах и архитектуре компьютеров, а также о будущих направлениях развития этих технологий.



Джим Ледин — основатель компании Ledin Engineering, Inc, специалист в сфере встраиваемых устройств, а также в разработке и тестировании программного обеспечения для них, сертифицированный эксперт по безопасности информационных систем и тестированию на проникновение.

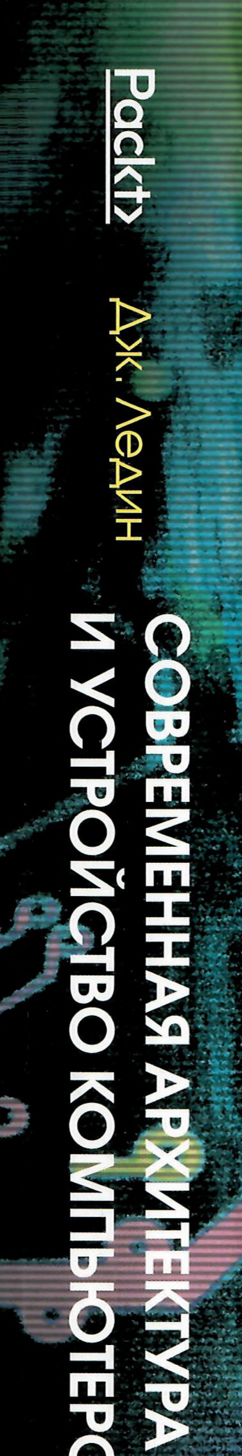


Цветные иллюстрации можно скачать по ссылке https://zip.bhv.ru/ 9785977518703.zip, а также со страницы книги на сайте bhv.ru.



191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru





bhv

Джим Ледин

СОВРЕМЕННАЯ АРХИТЕКТУРА И УСТРОЙСТВО КОМПЬЮТЕРОВ

Изучите архитектуры x86, ARM, RISC-V, устройство компьютеров, смартфонов и облачных серверов

2-е издание



Материалы на www.bhv.ru



Modern Computer Architecture and Organization

Second Edition

Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers

Jim Ledin



BIRMINGHAM—MUMBAI

Джим Ледин

СОВРЕМЕННАЯ АРХИТЕКТУРА И УСТРОЙСТВО КОМПЬЮТЕРОВ

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2024

УДК 004.43
ББК 32.973.26-018.1
Л39

Ледин Дж.

Л39 Современная архитектура и устройство компьютеров: Пер. с англ. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2024. — 656 с.: ил.

ISBN 978-5-9775-1870-3

Книга посвящена архитектуре и принципам работы современных цифровых устройств: компьютеров, смартфонов, облачных серверов. Подробно рассмотрены архитектуры процессоров и наборы инструкций x86, x64, ARM и RISC-V. Описано внутреннее устройство процессора, логические вентили, триггеры, регистры, конвейеры, показаны режимы адресации, обработка прерываний, представлены основы машинной логики, методы повышения вычислительной производительности процессоров. Приведен пример разработки процессора RISC-V на базе недорогой платы FPGA (ПЛИС). Описаны принципы виртуализации и технологии, лежащие в основе виртуальных машин, рассмотрены архитектурные решения для обеспечения кибербезопасности и конфиденциальности вычислений. Уделено внимание специализированным компьютерным архитектурам: облачным серверам, мобильным устройствам, процессорам для нейронных сетей и машинного обучения, блокчейна и майнинга, беспилотного транспорта. На практическом примере показаны квантовые вычисления, рассмотрены другие перспективные направления в вычислительных архитектурах.

Для программистов, инженеров и системных архитекторов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Дмитрия Мешкова*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Оформление обложки	*Зои Канторович*

Подписано в печать 05.09.23.
Формат 70×100 $^{1}/_{16}$. Печать офсетная. Усл. печ. л. 52,89.
Тираж 1500 экз. Заказ № 7966.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, Россия, г. Ульяновск, ул. Гончарова, 14

ISBN 978-1-80323-451-9 (англ.)
ISBN 978-5-9775-1870-3 (рус.)

Содержание

Предисловие

Я — разработчик программного обеспечения, а не инженер-электронщик. На протяжении своей карьеры я разрабатывал самое разное программное обеспечение для решения множества различных проблем. Однако по случайности или причуде судьбы значительная часть моей карьеры была связана с созданием программного обеспечения, существенно более близкого к аппаратным средствам, чем у многих, а может быть, и у большинства, разработчиков программ в наши дни.

В первые годы моего увлечения компьютерами я быстро понял, что невероятно несовершенные по сегодняшним меркам устройства, к которым у меня был доступ, не смогут решать интересные задачи, если я не научусь программировать их на ассемблере. Поэтому я научился писать программы на ассемблере Z80, а затем и на ассемблерах для процессоров 6502 и 80*x*86.

Программирование на ассемблере во многом отличается от написания программ на языках более высокого уровня. Тут вы чувствуете, как работает оборудование. Вы не можете игнорировать механизм распределения памяти, вам необходимо подстроить свой код под него. Вы не можете не принять во внимание имеющиеся в вашем распоряжении регистры, это ваши переменные, и вам нужно тщательно соблюдать строгий порядок в управлении их работой. Вы также учитесь налаживанию взаимодействия с другими устройствами через порты ввода-вывода, ведь по сути это единственный способ взаимодействия цифровых устройств друг с другом. Однажды, работая над особенно сложной задачей, я проснулся посреди ночи и понял, что мне снился ассемблер 80*x*86.

Моя карьера и, что более важно, имеющееся в моем распоряжении оборудование развивались. В то время я получил работу своей мечты в отделе исследований и разработок компании-производителя компьютеров. Моя задача заключалась в улучшении взаимодействия операционных систем с нашим оборудованием, и я создавал драйверы устройств, чтобы использовать уникальные функции наших персональных компьютеров. И в такой работе также было важно иметь хорошие прикладные знания о том, как работают аппаратные средства.

Разработка программного обеспечения развивалась. Языки, которые мы использовали, становились более абстрактными, операционные системы, виртуальные машины, контейнеры и общедоступная облачная инфраструктура все чаще скрывали

от нас, разработчиков программ, детали используемого оборудования. Недавно в социальной сети я общался с программистом на LISP, который не понимал, что в конечном счете его прекрасные функциональные декларативные структуры преобразуются в коды операций и значения в регистрах центрального процессора. Похоже, у него не было рабочей модели того, как работают компьютеры, на которую он смог бы положиться. Ему не требовалось в этом разбираться, но я думаю, что если бы он это сделал, то стал бы лучшим программистом, чем был.

В конце своей карьеры я работал над высокопроизводительными системами мирового класса. Перед коллективом, который я возглавлял, была поставлена задача создать одну из самых эффективных финансовых бирж в мире.

Для того чтобы сделать это, нам снова нужно было основательно разобраться в том, как работает оборудование нашей системы. Это позволило нам в полной мере воспользоваться впечатляющей производительностью, предлагаемой современными аппаратными средствами. В это время, чтобы попытаться описать наш подход, мы позаимствовали термин из автоспорта. В 1970-х годах лучшим гонщиком "Формулы-1" был Джеки Стюарт. В одном из интервью у него спросили: "Нужно ли обладать навыками инженера, чтобы стать великим гонщиком?" Джеки ответил: "Нет, но вы должны испытывать „механическую симпатию“ к автомобилю". По сути, вам необходимо хорошо понимать возможности используемого оборудования, чтобы максимально эффективно использовать все его преимущества.

Мы приняли эту идею "механической симпатии" и применили ее в нашей работе. Например, самые большие потери производительности в нашей торговой системе были связаны с отсутствием в кеш-памяти необходимых данных. Если данные, которые мы хотели обработать, отсутствовали в соответствующем кеше, когда они были необходимы, производительность нашей системы падала на порядки. Поэтому нам нужно было спроектировать наш код, даже если бы он был написан на языке высокого уровня и запущен на виртуальной машине, таким образом, чтобы добиться максимальной вероятности наличия нужных данных в кеше. Нам нужно было понять и научиться управлять параллелизмом в наших многоядерных процессорах, а также находить и извлекать выгоду из таких аспектов, как строки кеша процессора, преимущественно блочная структура памяти и других устройств хранения данных. Результатом стало достижение таких уровней производительности, которые некоторые люди считали невозможными. Современное оборудование впечатляет своими возможностями, когда вы используете их в полной мере.

Этот интерес к оборудованию связан не только с высокопроизводительными вычислениями. Несмотря на расхождение в конкретных оценках, все согласны с тем, что значительная часть углерода, вырабатываемого нами как биологическим видом, обусловлена энергоснабжением центров обработки данных, в которых хранится наш код. Я не могу представить себе ни одной области человеческой деятельности, которая была бы столь же неэффективной, как программное обеспечение, — для большинства систем вполне достижимо увеличение скорости до 100 раз, если вы

немного поработаете над управлением потоком информации через ваше оборудование. Почти для всех систем можно добиться тысячекратного увеличения скорости при определенной целенаправленной работе, но даже если бы мы могли получить хотя бы десятикратное ускорение за счет лучшего понимания того, как работает наш код и как он использует оборудование, на котором он работает, мы могли бы уменьшить углеродный след вычислений также в 10 раз. Эта идея гораздо важнее, чем повышение производительности лишь ради самой производительности.

В конце концов, есть определенная степень необходимого понимания того, как работает ваш компьютер, и есть риск потерять представление о том, как функционирует оборудование, от которого мы все зависим. Признаюсь, я фанат своего дела. Я люблю разбираться в том, как все работает. Не всем нужно досконально знать, как работает оборудование, но относиться к нему как к некой магии — плохая идея, потому что это не магия. Это инженерное дело, а инженерная практика всегда связана с компромиссами. Вы будете удивлены, как часто принципы работы вашего оборудования проявляют себя и влияют на поведение ваших программ, какими бы высокоуровневыми они ни были, даже если речь идет об написании кода для облачных систем на LISP.

Для таких, как я, второе издание книги Джима Ледина "Современная архитектура и организация компьютеров" — это восторг.

Я не инженер-электронщик и не хочу им быть. Однако для меня жизненно важной частью моих навыков как разработчика программного обеспечения является наличие хорошей практической модели того, как на самом деле работают аппаратные средства, на которые я полагаюсь. Я хочу поддерживать и развивать концепцию "механической симпатии".

Эта книга ведет нас от основных концепций вычислений через рассмотрение первых компьютеров и первых процессоров к потенциалу квантовых вычислений и другим направлениям ближайшего будущего, которые могут быть положены в основу нашей вычислительной техники. Возможно, вы захотите понять, как работают современные процессоры, и досконально разобраться с их поразительной эффективностью и их способностью поддерживать свою работу, извлекая нужные данные из хранилищ, которые в сотни раз медленнее их самих. Вас также могут заинтересовать сложные идеи, выходящие за рамки только лишь аппаратных средств, например о том, как работает майнинг криптовалют или как выглядит архитектура современного самоуправляемого автомобиля. Эта книга может ответить на эти и многие другие вопросы.

Я думаю, что не только специалисты по теории вычислений и инженеры по вычислительной технике, но и разработчики программного обеспечения смогут лучше выполнять свою работу, располагая некоторым представлением о том, как работают устройства, которые они используют в своей повседневной работе. Пытаясь разобраться с крупной и сложной проблемой в программном коде, я до сих пор часто думаю: "Стоп! Ведь на самом деле это всего лишь биты, байты и коды операций,

так что же здесь происходит?" Это похоже на то, как химик, разбираясь в молекулах и химических соединениях, может вернуться к основным принципам, чтобы решить какую-либо сложную задачу. Это реальные первоосновы, и они могут помочь всем нам лучше понимать их.

Я знаю, что буду регулярно обращаться к этой книге в течение многих лет, и надеюсь, что вам понравится делать то же самое.

Дэйв Фарли (Dave Farley),
независимый консультант по разработке программного обеспечения
и учредитель компании Continuous Delivery Ltd

Составители

Об авторе

Джим Ледин (Jim Ledin) — исполнительный директор компании Ledin Engineering, Inc. Джим является экспертом в области проектирования и тестирования встроенного программного обеспечения и аппаратных средств, а также экспертом в области оценки кибербезопасности систем и тестирования на проникновение. Джим имеет степень бакалавра в области аэрокосмической техники от Университета штата Айова и степень мастера в области электротехники и вычислительной техники от Технологического института Джорджии. Он является зарегистрированным профессиональным инженером-электриком в штате Калифорния, сертифицированным специалистом по безопасности информационных систем (Certified Information System Security Professional, CISSP), сертифицированным этичным хакером (Certified Ethical Hacker, CEH) и сертифицированным специалистом по тестированию на проникновение (Certified Penetration Tester, CPT).

Я хочу поблагодарить мою жену Линду и дочь Эмили за их терпение и поддержку в то время, когда я был сосредоточен на работе над этим проектом. Я люблю вас, мои дорогие!

Хочу также поблагодарить доктора Сару М. Нойвирт и Истока Йераса за их усердную работу по рецензированию каждой главы этой книги. Ваш вклад помог мне создать гораздо лучшую книгу!

Отдельная благодарность Дэйву Фарли за столь красноречивое предисловие.

О рецензентах

Д-р Сара М. Нойвирт (Dr. Sarah M. Neuwirth) — научный сотрудник с докторской степенью в группе модульных суперкомпьютеров и квантовых вычислений Университета Гёте (Франкфурт, Германия). Она также занимает должность приглашенного исследователя в Юлихском исследовательском центре суперкомпьютеров (Германия). Сара располагает более чем девятилетним опытом работы в академической сфере. В сферу ее исследовательских интересов входят высокопроизводительные системы хранения данных, параллельный ввод-вывод и файловые системы,

модульные суперкомпьютеры (разукрупнение ресурсов и виртуализация), стандартизированный сравнительный анализ кластеров, высокопроизводительные вычисления и сети, распределенные системы и протоколы связи.

Кроме того, Сара разработала учебную программу и в течение последних девяти лет вела курсы по параллельной компьютерной архитектуре, высокопроизводительным сетям с внутрисистемной коммутацией и распределенным системам. В 2018 г. Сара получила докторскую степень по теории вычислительных машин в Гейдельбергском университете (Германия). Она защитила ученую степень с отличием (summa cum laude) и была удостоена научной награды ZONTA 2019 за свою выдающуюся диссертацию. Сара также имеет степень магистра наук (2012) и степень бакалавра наук (2010) в области теории вычислений и математики Университета Мангейма (Германия). Она выступала в роли технического рецензента для нескольких престижных конференций и журналов, посвященных высокопроизводительным вычислениям, в том числе для серии конференций IEEE/ACM SC, ACM ICPP, IEEE IDPDS, IEEE HPCC, семинара PDSW в рамках IEEE/ACM SC, семинара PERMAVOST в рамках ACM HPDC, ACM TOCS, IEEE Access и Elsevier FGCS.

Исток Йерас (Iztok Jeras) получил степень бакалавра электротехники и степень магистра по теории вычислительных машин в Люблянском университете. Он работал в нескольких словенских компаниях над микроконтроллерами, решениями на основе программируемых логических интегральных схем (field-programmable gate array, FPGA) и специализированных интегральных схем (application-specific integrated circuit, ASIC), а также со встроенным программным обеспечением и операционной системой Linux. В свободное время Исток исследует клеточные автоматы и участвует в проектах разработки цифровых систем с открытым исходным кодом. В последнее время он сосредоточился на архитектуре набора инструкций RISC-V.

Присоединяйтесь к нашему сообществу в Discord!

Присоединяйтесь к сообществу этой книги в Discord для участия в ежемесячных авторских семинарах "Спроси о чем угодно":

https://discord.gg/7h8aNRhRuY



Вступление

Приветствую читателей второго издания книги "Современная архитектура и организация компьютеров". Мне было очень приятно получить множество отзывов и комментариев от читателей первого издания. Я ценю любые отзывы своих читателей, особенно тех, кто обращает мое внимание на какие-либо ошибки или упущения.

В этой книге представлены основные технологии и компоненты, используемые в архитектурах современных процессоров и компьютеров, и обсуждается, как различные архитектурные решения помогают создавать конфигурации компьютеров, оптимизированные для реализации конкретных потребностей.

Сильно упрощая ситуацию, можно сказать, что современные компьютеры — это сложные устройства. Тем не менее, если применить для их рассмотрения иерархический принцип, то становятся ясными функции каждого уровня сложности. В главах этой книги мы затронем очень много важных тем, и у нас будет возможность лишь ограниченного ознакомления с каждой из них. Моя цель состоит в том, чтобы предоставить ясное введение в каждую важную технологию и подсистему, которые вы можете найти в современном вычислительном устройстве, и объяснить их связь с другими компонентами системы.

Это издание содержит обновленную информацию о технологиях, которые продвинулись вперед со времени публикации первого издания, в него также добавлен значительный объем новых сведений по нескольким важным областям, связанным с архитектурой компьютеров. Новые главы охватывают вычислительные архитектуры для кибербезопасности, блокчейна и майнинга биткоинов, а также самоуправляемых транспортных средств.

Безопасность вычислительных систем всегда была важным вопросом, но недавние случаи использования серьезных уязвимостей в широко распространенных операционных системах и приложениях привели к значительным негативным последствиям, которые ощущаются в странах по всему миру. Эти кибератаки подчеркнули необходимость включения средств защиты от киберугроз в качестве основополагающего элемента архитектуры вычислительных систем.

Я не буду приводить длинный список ссылок для дальнейшего чтения. Интернет — ваш лучший помощник в этом отношении.

Если вам удастся обойти суету политической и социальной аргументации в Интернете, вы окажетесь в просторной, прохладной и тихой библиотеке, содержащей ог-

ромное количество накопленных человеческих знаний. Научитесь использовать расширенные функции вашей любимой поисковой системы. Кроме того, сможете отличать качественную информацию от мнений неосведомленных людей. Если у вас есть какие-либо сомнения относительно информации, которую вы ищете, проверьте несколько источников. Обратите внимание на источник: если вам нужна информация о процессоре Intel, ищите документацию, опубликованную Intel.

К концу этой книги у вас сложится четкое представление о компьютерных архитектурах, используемых в настоящее время в самых разных цифровых системах. Кроме того, вы сможете расширить свое представление о соответствующих тенденциях в архитектурных технологиях, разрабатываемых в настоящее время, а также о некоторых возможных прорывных достижениях ближайшего будущего, которые могут кардинально повлиять на развитие архитектуры вычислительных систем.

Для кого эта книга

Эта книга предназначена для разработчиков программного обеспечения, студентов, изучающих вычислительную технику, проектировщиков систем, специалистов в области теории вычислительных машин и инженерного анализа, а также всех, кто хочет понять архитектуру и принципы проектирования, лежащие в основе современных компьютерных систем любого типа — от миниатюрных встроенных устройств до смартфонов и ферм облачных серверов размером с большой склад. Читатели также смогут ознакомиться с тенденциями развития этих технологий в ближайшие годы. Общее понимание компьютерных процессоров полезно, но не обязательно.

Как организована эта книга

Глава 1 "Введение в архитектуру компьютеров" начинается с краткой истории автоматических вычислительных устройств и описывает значительные технологические достижения, которые привели к ряду прорывов в вычислительных возможностях. Затем следует обсуждение закона Мура с оценкой его применимости в предыдущие десятилетия и возможных последствий для будущего. Основные понятия компьютерной архитектуры представлены в контексте микропроцессора 6502.

Глава 2 "Цифровая логика" представляет транзисторы, используемые в качестве переключающих устройств, и объясняет их использование в построении логических вентилей. Затем мы увидим, как создаются триггеры и регистры путем объединения простых вентилей. Здесь вводится понятие последовательностной логики, т. е. логики, содержащей информацию о состоянии. Глава заканчивается обсуждением цифровых схем с синхронизацией.

Глава 3 "Элементы процессора" начинается с концептуального описания универсального процессора. Мы рассмотрим концепции набора инструкций, набора реги-

стров, а также загрузки, декодирования, выполнения и определения последовательности инструкций.

Также здесь обсуждаются операции загрузки и сохранения данных в памяти. Глава включает в себя описание инструкций ветвления и их использования в циклах и условной обработке. Представлены некоторые практические соображения, приводящие к необходимости обработки прерываний и операций ввода-вывода.

В *главе 4 "Компоненты компьютерной системы"* обсуждаются память компьютера и ее интерфейс с процессором, в том числе многоуровневое кеширование. Описаны требования к вводу-выводу, включая обработку прерываний, буферизацию и выделенные процессоры ввода-вывода. Мы обсудим некоторые особые требования к устройствам ввода-вывода, включая клавиатуру и мышь, дисплей и сетевой интерфейс. Глава заканчивается описательными примерами этих компонентов в современных компьютерных системах, среди которых мобильные устройства, персональные компьютеры, игровые системы, облачные серверы и специализированные системы машинного обучения.

В *главе 5 "Аппаратно-программный интерфейс"* обсуждается реализация высокоуровневых функций, которые должна предоставлять операционная система компьютера, включая ввод-вывод для накопителей, обмен данными по сетям и взаимодействие с пользователями. В этой главе описываются уровни программного обеспечения, реализующие эти функции, начиная с уровня набора инструкций и регистров процессора. Также описаны функции операционной системы, в том числе ее загрузка, многопроцессорность и многопоточность.

Глава 6 "Специализированные вычисления" исследует области вычислений, которые, как правило, менее очевидны для большинства пользователей, включая системы реального времени, цифровую обработку сигналов и обработку данных в графических процессорах. Мы обсудим уникальные требования, связанные с каждой из этих областей, и рассмотрим примеры современных устройств, реализующих эти функции.

Глава 7 "Архитектура процессора и памяти" подробно рассматривает архитектурные решения современных процессоров, включая фон-неймановскую, гарвардскую и модифицированную гарвардскую. Здесь обсуждается реализация виртуальной памяти со страничной организацией. Представлена практическая реализация функций управления памятью в архитектуре компьютера, а также функций блока управления памятью.

В *главе 8 "Методы повышения производительности"* обсуждается ряд методов повышения производительности, которые обычно используются для достижения максимальной скорости выполнения программ в реальных компьютерных системах. В этой главе рассматриваются наиболее важные методы повышения производительности систем, включая использование кеш-памяти, конвейерной обработки инструкций, параллелизма инструкций и обработку на основе принципа "одна инструкция, множество данных" (single instruction-multiple data SIMD).

Глава 9 "Специализированные расширения процессоров" фокусируется на расширениях, обычно реализуемых на уровне набора инструкций процессора, с целью предоставления дополнительных системных возможностей помимо общих требований к обработке данных. Представленные расширения включают в себя привилегированные режимы процессора, арифметику с плавающей запятой, управление питанием и безопасностью системы.

Глава 10 "Современные архитектуры и наборы инструкций процессоров" исследует особенности архитектуры и наборов инструкций современных процессоров, включая *x*86, *x*64 и ARM. Одной из проблем, возникающих при производстве семейства процессоров в течение нескольких десятилетий, является необходимость поддерживать обратную совместимость с кодом, написанным для процессоров прежних поколений. Необходимость поддержки унаследованного кода приводит к увеличению сложности процессоров более поздних поколений. В этой главе рассмотрены некоторые атрибуты этих процессорных архитектур, которые обусловлены поддержкой устаревших требований.

Глава 11 "Архитектура и набор инструкций RISC-V" представляет захватывающую новую архитектуру процессоров RISC-V (произносится как "риск файв") и соответствующий набор инструкций. RISC-V — это свободно распространяемая спецификация с полностью открытым исходным кодом для вычислительной архитектуры с сокращенным набором инструкций. Выпущена полная спецификация набора инструкций, и в настоящее время доступно несколько аппаратных реализаций этой архитектуры. Продолжается разработка спецификаций для ряда расширений набора инструкций. В этой главе рассматриваются функции, доступные в архитектуре RISC-V, варианты ее реализации и представлен набор инструкций RISC-V. Мы также обсудим применение архитектуры RISC-V в мобильных устройствах, персональных компьютерах и серверах.

Глава 12 "Виртуализация процессоров" представляет концепции, связанные с виртуализацией процессоров, и объясняет многочисленные преимущества, возникающие в результате применения виртуализации. Глава включает примеры виртуализации на основе инструментов и операционных систем с открытым исходным кодом. Эти инструменты позволяют создавать на обычном компьютере точные с точки зрения набора инструкций представления различных компьютерных архитектур и операционных систем. Мы также обсудим преимущества виртуализации при разработке и развертывании реальных программных приложений.

Глава 13 "Специализированные компьютерные архитектуры" объединяет темы, рассмотренные в предыдущих главах, в целях выработки подхода к созданию архитектуры компьютерной системы для выполнения уникальных требований пользователя. Мы обсудим конкретные категории областей применения, включая мобильные устройства, персональные компьютеры, игровые системы, поисковые системы и нейронные сети.

Глава 14 "Архитектуры для обеспечения кибербезопасности и конфиденциальности вычислений" сосредоточена на потребностях в защите критически важных об-

ластей, включая системы национальной безопасности и обработку финансовых транзакций. Эти системы должны быть устойчивыми к широкому спектру угроз кибербезопасности, в том числе к проникновению вредоносного кода, атакам через скрытые каналы и путем физического доступа к аппаратным средствам компьютеров. Среди тем, рассматриваемых в этой главе, — угрозы кибербезопасности, шифрование, цифровые подписи и архитектурные решения для защиты аппаратных средств и программного обеспечения.

Взрыв интереса к криптовалютам и их растущее признание основными финансовыми институтами и торговыми предприятиями демонстрируют, что данная область вычислительной техники находится на пути непрерывного роста. В это издание книги добавлена глава о блокчейне и вычислительных требованиях майнинга биткоинов.

Глава 15 "Архитектуры блокчейна и майнинга биткоинов" представляет концепции, связанные с блокчейном, открытым, криптографически защищенным реестром, содержащим последовательность транзакций. За введением следует обзор процесса майнинга биткоинов, который добавляет транзакции в блокчейн Bitcoin и вознаграждает тех, кто выполняет эту задачу, оплатой в биткоинах. Для обработки биткоинов требуется высокопроизводительное вычислительное оборудование, которое представлено с точки зрения компьютерной архитектуры для майнинга биткоинов текущего поколения.

Продолжающийся рост числа автомобилей с возможностью частичного или полного автономного вождения требует надежных вычислительных систем с высокой производительностью, отвечающих требованиям безопасной автономной эксплуатации транспортных средств на дорогах общего пользования.

Глава 16 "Архитектуры для самоуправляемых автомобилей" представляет возможности, которые должны быть реализованы в вычислительных архитектурах самоуправляемых автомобилей. Глава начинается с обсуждения требований по обеспечению безопасности самоуправляемого автомобиля и его пассажиров, а также других транспортных средств, пешеходов и стационарных объектов. Затем мы рассмотрим типы датчиков и источники информации, которые предоставляют самоуправляемым автомобилям входные данные во время движения, а также типы вычислений, необходимых для эффективного управления автомобилем. Глава завершится обзором примера архитектуры компьютера для самоуправляемого автомобиля.

В *главе 17 "Квантовые вычисления и другие перспективные направления в вычислительных архитектурах"* оцениваются перспективы развития компьютерных архитектур. В ней рассматриваются важные достижения и современные тенденции, которые привели к текущему состоянию компьютерных архитектур, а также прогнозируется дальнейшее развитие этих тенденций для определения возможных будущих технологических направлений. Также обсуждаются потенциально прорывные технологии, которые могут изменить направление развития компьютерных архитектур в будущем. В заключение я дам рекомендации по некоторым подходам к профессиональному совершенствованию архитектора компьютеров, которые могут помочь сформировать набор полезных для будущего навыков.

Как и прочие главы, каждая из трех новых глав содержит упражнения, помогающие лучше понять тему главы и закрепить полученную информацию в вашей базе знаний.

Я надеюсь, что вы получите удовольствие от прочтения обновленного издания, так же, как и я, когда работал над этой книгой. Приятного чтения!

Как получить максимальную отдачу от этой книги

Каждая глава этой книги завершается набором упражнений. Для того чтобы извлечь из книги максимальную пользу и закрепить в уме некоторые из наиболее сложных концепций, я рекомендую вам поработать над каждым упражнением. Полные решения для всех упражнений приведены в книге и доступны в Интернете по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

В случае необходимости внесения изменений в примеры кода и ответы к упражнениям эти изменения появятся в этом же репозитории GitHub.

Загрузите файлы с примерами кода

Комплект исходного кода для данной книги размещен на GitHub по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**. У нас также есть другие комплекты исходного кода из нашего обширного каталога книг и видео, доступных по адресу **https://github.com/PacktPublishing/**. Рекомендуем ознакомиться с ними!

Загрузите цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями снимков экрана и диаграмм, использованных в этой книге. Вы можете скачать этот файл по адресу: **https://static.packt-cdn.com/downloads/9781803234519_ColorImages.pdf**[1].

Условные обозначения и соглашения

В книге используются следующие типографские условные обозначения.

`CodeInText` указывает на примеры кода в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути, пользовательский ввод и имена учетных записей в Twitter. Пример: "Вычитание с использованием инструкции `SBC` может немного сбить с толку начинающих программистов на языке ассемблера 6502".

[1] Файл с цветными иллюстрациями для русскоязычного издания представлен здесь: **https://zip.bhv.ru/9785977518703.zip**.

Листинг кода обозначается следующим образом:

```
; Сложение четырех байтов с использованием режима непосредственной адресации
LDA #$04
CLC
ADC #$03
ADC #$02
ADC #$01
```

Любые примеры ввода или вывода с использованием командной строки записываются следующим образом:

```
C:\>bcdedit
Windows Boot Manager
--------------------
identifier {bootmgr}
```

Полужирный шрифт обозначает новый термин или важное слово, интернет-адрес и адрес электронной почты; таким же образом выделяются в тексте слова, которые вы видите на экране, — названия пунктов меню или элементов диалоговых окон. Пример: "Поскольку теперь существуют четыре набора, поле **Набор** в физическом адресе уменьшается до двух битов, а поле **Тег** увеличивается до 24 бит".



Предупреждения или важные примечания выглядят таким образом.

Свяжитесь с нами

Обратная связь со стороны наших читателей всегда приветствуется.

Общие отзывы. Отзывы общего характера отправляйте по электронной почте **feedback@packtpub.com**, указав в теме сообщения название книги. Если у вас есть вопросы по любому аспекту этой книги, пожалуйста, напишите нам по адресу **questions@ packtpub.com**.

Опечатки. Мы приложили все разумные усилия, чтобы обеспечить точность содержимого книги, но ошибки иногда случаются. Если вы нашли опечатку в этой книге, мы будем признательны, если вы сообщите нам об этом. Зайдите на сайт **http://www.packtpub.com/submit-errata**, выберите свою книгу, щелкните по ссылке **Errata Submission Form** (Форма уведомления об опечатках) и введите сведения об опечатке.

Пиратство. Если в Интернете вам встретятся незаконные копии наших произведений в любой форме, мы будем признательны, если вы сообщите нам адрес их местонахождения или название веб-сайта. Пожалуйста, отправьте нам ссылку на такой материал по адресу **copyright@packtpub.com**.

Если вы хотите стать автором. Если есть тема, в которой у вас есть опыт, и вы заинтересованы в написании или участии в написании книги, пожалуйста, посетите веб-сайт **http://authors.packtpub.com**.

Поделитесь своими мыслями

Мы будем рады узнать ваше мнение после того, как вы прочтете второе издание книги "Современная архитектура и организация компьютеров". Отсканируйте приведенный ниже QR-код, чтобы перейти на страницу рецензирования этой книги и поделиться своим мнением.



Ваш отзыв важен для нас и технического сообщества, он поможет нам убедиться, что мы предоставляем информацию наивысшего качества.

1

Введение в архитектуру компьютеров

Архитектура автоматических вычислительных систем прошла долгий путь от первых механических вычислителей, созданных почти два века назад, до широкого спектра современных электронных компьютерных технологий, которые мы прямо или косвенно используем каждый день. На этом пути были периоды постепенных технологических улучшений, чередующиеся с прорывными достижениями, которые резко меняли траекторию развития отрасли. Можно ожидать, что эти тенденции сохранятся и в ближайшие годы.

В 1980-х годах, на заре развития персональных вычислительных устройств, студентам и техническим специалистам, стремившимся узнать как можно больше о компьютерных технологиях, был доступен лишь ограниченный набор предметов для изучения. Если они располагали собственным компьютером, то это, скорее всего, был IBM PC или Apple II. Если они работали в организации с вычислительным центром, то могли бы использовать большую ЭВМ от IBM или мини-компьютер VAX от Digital Equipment Corporation. Эти примеры и ограниченное число подобных систем охватывали практически все возможности знакомства большинства людей с компьютерными системами того времени.

Сегодня существует множество специализированных вычислительных архитектур для удовлетворения самых разных потребностей пользователей. В своих карманах мы носим миниатюрные компьютеры, которые могут совершать телефонные вызовы, записывать видео и предоставлять полнофункциональный доступ в Интернет. Персональные компьютеры (ПК) остаются популярными в формате, внешне похожем на ПК прошлых десятилетий. Однако современные ПК на несколько порядков превосходят предыдущие поколения по вычислительной мощности, объему памяти, дисковому пространству, графической производительности и доступным способам связи. Эти возможности позволяют современным ПК легко решать задачи, выпол-

нение которых было бы немыслимо на ранних ПК, такие как формирование трехмерных изображений с высоким разрешением в реальном времени.

Компании, предлагающие веб-сервисы сотням миллионов пользователей, создают огромные хранилища, заполненные тысячами тесно скоординированных компьютерных систем, способных реагировать на постоянный поток пользовательских запросов с необычайной скоростью и точностью. Системы машинного обучения обучаются на основе анализа огромных объемов данных для выполнения сложных действий, таких как вождение автомобилей.

Эта глава начинается с представления некоторых вычислительных устройств, сыгравших важную роль в истории отрасли, и связанных с ними технологических прорывов. Затем мы рассмотрим некоторые важные современные тенденции, связанные с технологическими достижениями, и познакомимся с основными концепциями компьютерной архитектуры, включая внимательное изучение микропроцессора 6502 и его набора инструкций. В этой главе будут рассмотрены следующие темы:

- эволюция автоматических вычислительных устройств;
- закон Мура;
- архитектура компьютеров.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Эволюция автоматических вычислительных устройств

В этом разделе рассматриваются некоторые классические машины из истории автоматических вычислительных устройств, при этом основное внимание уделяется важным достижениям, воплощенным в каждой из них. Аналитическая машина Бэббиджа включена сюда благодаря множеству гениальных нововведений, представленных в ее конструкции. Обсуждение других систем обусловлено тем, что они воплощали в себе значительные технологические достижения и за время своего существования выполнили значительную практическую работу.

Аналитическая машина Чарльза Бэббиджа

Несмотря на то что рабочая модель этой **аналитической машины** никогда не была построена, в подробных записях, которые вел **Чарльз Бэббидж** с 1834 г. до своей смерти в 1871 г., была описана вычислительная архитектура, которая казалась вполне завершенной и работоспособной. Эта аналитическая машина предназначалась для использования в качестве универсального программируемого вычисли-

тельного устройства. Ее конструкция была полностью механической, а изготовить машину предполагалось по большей части из латуни. Аналитическая машина должна была приводиться в движение валом от парового двигателя.

Для своей аналитической машины Бэббидж позаимствовал перфокарты ткацкого станка Жаккара, вращающиеся цилиндры со штифтами, используемые в музыкальных шкатулках, и технологии своей более ранней разностной машины (также не законченной при его жизни и представлявшей собой скорее специализированное счетное устройство, чем компьютер), в остальном ее конструкция была оригинальным творением Бэббиджа.

В отличие от большинства современных компьютеров, эта аналитическая машина представляла числа в десятичном виде со знаком. Решение использовать числа с основанием 10, а не логику большинства современных компьютеров с основанием 2, было обусловлено фундаментальным различием между механическими технологиями и цифровой электроникой. Изготовление механических колес с 10 положениями — несложная задача, поэтому Бэббидж выбрал привычный человеку формат с основанием 10, т. к. он был ненамного сложнее с технической точки зрения, чем использование какой-либо другой системы счисления. С другой стороны, простые цифровые схемы не способны поддерживать 10 различных состояний с той же легкостью, с которой это достигается с помощью механического колеса.

Все числа в аналитической машине состояли из 40 десятичных цифр. Большое количество цифр, вероятно, было выбрано для облегчения решения проблем с числовым переполнением. Эта аналитическая машина не поддерживала математику с плавающей запятой.

Каждое число хранилось на вертикальной оси, содержащей 40 колес, каждое из которых могло находиться в 10 положениях, соответствующих цифрам от 0 до 9. Сорок первое колесо определяло знак: любое четное число на этом колесе представляло положительный знак, а любое нечетное число — отрицательный. Ось аналитической машины была в чем-то аналогична регистру, используемому в современных процессорах, за исключением того, что считывание числа оси было разрушительным — после этой операции число обращалось в 0. Если требовалось сохранить значение оси после считывания, другая ось должна была сохранить копию значения во время считывания. Числа переносились с одной оси на другую или использовались в вычислениях путем зацепления шестерни с каждым колесом и вращения колеса для извлечения числового значения. Набор осей, служащих системной памятью, назывался *складом*.

Для сложения двух чисел использовался процесс, напоминающий метод сложения, которому обучают школьников. Предположим, что число, хранящееся на одной оси, назовем его слагаемым, должно быть добавлено к числу на другой оси, которое мы назовем аккумулятором. Машина должна была соединить колесо каждой цифры слагаемого с соответствующим колесом цифры аккумулятора посредством зубчатой передачи. Затем она должна была одновременно вращать колесо каждой цифры слагаемого в направлении нуля, в то время как колесо цифры аккумулятора совершало эквивалентный поворот в возрастающем направлении. Если цифра ак-

кумулятора совершала переход от 9 к 0, следующая по старшинству цифра аккумулятора увеличивалась на 1. Эта операция *переноса* могла распространяться на столько цифр, сколько было необходимо (подумайте о добавлении 1 к 999 999). К концу процесса ось слагаемого должна была содержать значение 0, а ось аккумулятора — сумму двух чисел. Распространение переносов от одной цифры к другой было наиболее сложной с точки зрения механики частью процесса сложения.

Операции в аналитической машине выполнялись с помощью вращающихся цилиндров, похожих на барабан музыкальной шкатулки, в конструкции, называемой **мельницей**, которая аналогична блоку управления современного процессора.

Каждая инструкция аналитической машины была закодирована в вертикальном ряду позиций на цилиндре, где наличие или отсутствие штифта в определенном месте либо задействовало часть механизма машины, либо оставляло состояние соответствующей секции неизменным. Согласно оценке скорости выполнения операций, сделанной Бэббиджем, сложение двух 40-значных чисел, включая распространение переносов, должно было занять около 3 секунд.

Бэббидж разработал для своей машины несколько важных концепций, которые остаются актуальными и для современных компьютерных систем. Его конструкция поддерживала определенную степень параллельной обработки, предусматривающей одновременное выполнение операций умножения и сложения, что ускоряло вычисление серий значений, предназначенных для вывода в виде числовых таблиц. Математические операции, такие как сложение, поддерживали некоторую форму конвейерной обработки, при которой последовательные операции над разными значениями перекрываются во времени.

Бэббидж хорошо знал о трудностях, связанных со сложными механическими устройствами, таких как трение, люфт шестерен и износ с течением времени. Для того чтобы предотвратить ошибки, вызванные этими эффектами, в машину были встроены механизмы, называемые **блокировками**, которые применялись при передаче данных между осями. Блокировки заставляли колеса цифр занимать правильные положения и предотвращали накопление небольших ошибок, вызванных постепенным смещением колес в направлении неправильных значений. Использование блокировок аналогично усилению потенциально слабых входных сигналов для получения более сильных выходных сигналов с помощью цифровых логических вентилей в современных процессорах.

Для программирования аналитической машины предполагалось использовать перфокарты. Кроме того, в ней должны были поддерживаться операции ветвления и вложенные циклы. Самая сложная программа, предназначенная для выполнения на этой аналитической машине, была разработана Адой Лавлейс для вычисления чисел Бернулли — важной последовательности в теории чисел. Код машины для выполнения этих вычислений признан первой опубликованной компьютерной программой значительной сложности.

Бэббидж построил пробную модель части мельницы своей аналитической машины, которая в настоящее время выставлена в Музее науки в Лондоне.

ENIAC

Создание машины **ENIAC**, название которой расшифровывается как **Electronic Numerical Integrator and Computer** (электронная счетная машина-интегратор), было завершено в 1945 г., и она стала первым программируемым универсальным электронным компьютером. Система потребляла 150 кВт электроэнергии, занимала 170 квадратных метров площади и весила 27 тонн.

Ее конструкция была основана на электронных лампах, диодах и реле. Машина ENIAC содержала более 17 000 электронных ламп, которые выполняли функции переключающих элементов.

Подобно аналитической машине Бэббиджа, она использовала десятичное представление 10-значных десятичных чисел, реализованное с помощью 10-позиционных кольцевых счетчиков (кольцевой счетчик будет обсуждаться в *главе 2*).

Входные данные вводились с помощью устройства чтения перфокарт от IBM, а результаты вычислений выводились посредством перфорационной машины.

Архитектура ENIAC была способна выполнять сложные последовательности шагов обработки, включая циклы, ветвление и подпрограммы. Система содержала 20 десятиразрядных аккумуляторов, которые действовали как регистры современных компьютеров. Изначально у машины не было никакой памяти, кроме аккумуляторов. Если промежуточные значения требовались для использования в последующих вычислениях, эти данные приходилось записывать на перфокарты и по мере необходимости считывать обратно в машину. ENIAC могла выполнять около 385 операций умножения в секунду.

Программы ENIAC представляли собой проводные соединения на коммутационной панели и таблицы функций на основе переключателей. Программирование системы было трудоемким процессом, на выполнение которого у группы талантливых женщин-программистов часто уходили недели. Надежность была проблемой, т. к. вакуумные лампы регулярно выходили из строя, что требовало ежедневного выполнения процедуры поиска и устранения неполадок для выявления и замены неисправных ламп.

В 1948 г. ENIAC была улучшена за счет добавления возможности программирования системы с помощью перфокарт вместо коммутационных панелей. Это значительно ускорило разработку программ. Выступая в качестве консультанта по этому обновлению, Джон фон Нейман предложил архитектуру обработки, основанную на единой области памяти, содержащей программные инструкции и данные, блоке обработки с арифметико-логическим устройством и регистрами, а также блоке управления, содержащем регистр инструкций и программный счетчик. Многие современные процессоры наследуют эту общую структуру, известную теперь как **архитектура фон Неймана**. Мы подробно обсудим ее в *главе 3*.

Ранние области применения машины ENIAC включали в себя анализ, связанный с разработкой водородной бомбы, и расчет таблиц стрельбы для дальнобойной артиллерии.

IBM PC

За годы, прошедшие после создания ENIAC, произошло несколько технологических прорывов, которые привели к значительному прогрессу в компьютерной архитектуре.

- Появление транзистора, который изобрели в 1947 г. Джон Бардин, Уолтер Браттейн и Уильям Шокли, обеспечило значительное улучшение по сравнению с технологией электронных ламп, преобладавшей в то время. Транзисторы были быстрее, компактнее, потребляли меньше энергии, а после оптимизации производственных процессов стали намного надежнее ламп, подверженных отказам.
- После появления в 1958 г. идеи интегральных схем, предложенной Джеком Килби из Texas Instruments, начался процесс объединения большого количества ранее отдельных компонентов на одном кремниевом чипе.
- В 1971 г. компания Intel начала производство первого поступившего в открытую продажу микропроцессора Intel 4004. Модель 4004 предназначалась для использования в электронных калькуляторах и была специализирована для работы с 4-битными двоично-десятичными числами.

Со скромного начала в виде Intel 4004 микропроцессорная технология быстро развивалась в течение следующего десятилетия: число элементов схем на каждом чипе стремительно росло, а возможности микропроцессоров, реализованных на этих чипах, расширялись.

Микропроцессор Intel 8088

IBM выпустила персональный компьютер IBM PC в 1981 г. Оригинальный ПК содержал микропроцессор Intel 8088, работавший на тактовой частоте 4,77 МГц, и имел 16 Кбайт **оперативной памяти** (оперативное запоминающее устройство, ОЗУ) с возможностью расширения до 256 Кбайт. Он комплектовался одним или (по дополнительному запросу) двумя дисководами для гибких дисков. Также был доступен цветной монитор. Более поздние версии этого ПК поддерживали больше памяти, но поскольку часть адресного пространства была зарезервирована для видеопамяти и **постоянной памяти** (постоянное запоминающее устройство, ПЗУ), эта архитектура могла поддерживать ОЗУ максимальным объемом 640 Кбайт.

Процессор 8088 содержал четырнадцать 16-разрядных регистров. Четыре из них выполняли функции регистров общего назначения (AX, BX, CX и DX). Четыре других были регистрами сегментов памяти (CS, DS, SS и ES), которые расширяли адресное пространство до 20 бит. Сегментная адресация функционировала путем добавления 16-битного значения сегментного регистра, сдвинутого влево на 4 бита, к 16-битному значению смещения, содержащемуся в инструкции, для получения адреса физической памяти в диапазоне до 1 Мбайт.

Остальными регистрами процессора 8088 были **указатель стека** (SP), **указатель базы** (BP), **индекс источника** (SI), **индекс приемника** (DI), **указатель инструк-**

ций (IP) и **флаги состояния** (FLAGS). В современных процессорах семейства *x*86 реализована архитектура, удивительно похожая на этот набор регистров (детали архитектуры *x*86 описаны в *главе 10*). Наиболее очевидные различия между процессорами 8088 и *x*86 — увеличение разрядности регистров в *x*86 до 32 битов и добавление пары сегментных регистров (FS и GS), которые сегодня используются в основном как указатели данных в многопоточных операционных системах.

Процессор 8088 имел 8-разрядную внешнюю шину данных — это означало, что для чтения или записи 16-битного значения требовались два цикла шины. Это был шаг назад по производительности в сравнении с более ранним процессором 8086, который оснащался 16-разрядной внешней шиной. Однако применение 8-разрядной шины позволило сэкономить на производстве ПК и обеспечило совместимость с более дешевыми 8-разрядными периферийными устройствами. Такой экономичный подход к проектированию помог снизить цену ПК до уровня, доступного большему количеству потенциальных покупателей.

Для памяти программ и памяти данных совместно использовалось одно и то же адресное пространство, а доступ к памяти 8088 осуществлялся по одной шине. Другими словами, в 8088 была реализована архитектура фон Неймана. Набор инструкций 8088 включал в себя инструкции для перемещения данных, арифметических и логических операций, манипулирования строками, передачи управления (условные и безусловные переходы, а также вызов подпрограмм и возврат из них), операций ввода-вывода и различных дополнительных функций. Для выполнения одной инструкции процессору требовалось в среднем около 15 тактов, что обеспечивало скорость выполнения около 0,3 **млн инструкций в секунду** (million instructions per second, MIPS).

Процессор 8088 поддерживал девять различных режимов адресации памяти. Такое разнообразие режимов было необходимо для эффективного доступа к одному элементу за раз, а также для перебора последовательностей данных.

Сегментные регистры в архитектуре 8088 предоставляли, казалось бы, умный способ расширения диапазона адресуемой памяти без увеличения длины большинства инструкций, ссылающихся на ячейки памяти. Каждый сегментный регистр обеспечивал доступ к блоку памяти размером 64 Кбайт, начинающемуся с адреса физической памяти, кратного 16 байтам. Другими словами, 16-разрядный сегментный регистр представлял 20-битный базовый адрес с четырьмя младшими битами, равными нулю. Таким образом, инструкции могли ссылаться на любую ячейку в сегменте длиной 64 Кбайт, используя 16-битное смещение от адреса, определенного регистром сегмента.

Регистр CS устанавливал местоположение сегмента кода в памяти и использовался при извлечении инструкций и выполнении переходов, вызовов подпрограмм и возвратов из них. Регистр DS определял местоположение сегмента данных для использования инструкциями, перемещающими данные в память и из памяти. Регистр SS задавал местоположение сегмента стека, которое использовалось для выделения локальной памяти внутри подпрограмм и для хранения адресов возврата из подпрограмм.

Программы, которым требовалось менее 64 Кбайт в каждом из сегментов кода, данных и стека, могли полностью игнорировать сегментные регистры, поскольку эти регистры могли быть установлены один раз при запуске программы (компиляторы языков программирования делают это автоматически) и оставаться неизменными во время ее выполнения. Очень просто!

Все становилось немного сложнее, когда размер данных программы превышал 64 Кбайт. Применение сегментных регистров позволило создать стройное аппаратное решение, но их использование доставляло много проблем разработчикам программного обеспечения. Компиляторы для архитектуры 8088 различали указатели типа *near* и *far*. Указатель типа *near* представлял собой 16-битное смещение от базового адреса текущего сегментного регистра. Указатель типа *far* содержал 32 бита адресной информации: 16-битное значение регистра сегмента и 16-битное смещение. Указатели типа *far* потребляли дополнительные 16 бит памяти данных, а также требовали дополнительного времени на обработку.

Для одной операции доступа к памяти с использованием указателя типа *far* требовалось выполнить следующие шаги:

1. Сохранить текущее содержимое сегментного регистра во временной ячейке памяти.
2. Загрузить новое значение сегмента в регистр.
3. Выполнить операцию доступа к данным (чтение или запись, по мере необходимости) с использованием смещения от базового адреса сегмента.
4. Восстановить исходное значение сегментного регистра.

При использовании указателей типа *far* можно было объявлять объекты данных (например, массив символов, представляющих документ в текстовом редакторе) размером до 64 Кбайт. Если требовалась более крупная структура, нужно было придумать, как разбить ее на фрагменты размером не более 64 Кбайт и управлять ими самостоятельно. В результате таких манипуляций с сегментным регистром программы, которые требовали широкого доступа к элементам данных размером более 64 Кбайт, становились довольно сложными, что вело к увеличению размера кода и замедлению его выполнения.

На материнской плате IBM PC располагался разъем для предлагаемого отдельно сопроцессора Intel 8087 для вычислений с плавающей запятой. Разработчики 8087 изобрели форматы данных и правила обработки для 32- и 64-разрядных чисел с плавающей запятой, которые были закреплены в 1985 г. в виде стандарта IEEE 754 для вычислений с плавающей запятой — этот стандарт находит практически универсальное применение и сегодня. Сопроцессор 8087 может выполнять около 50 000 операций с плавающей запятой в секунду. Мы подробно рассмотрим их в *главе 9*.

Микропроцессоры Intel 80286 и 80386

Второе поколение компьютеров IBM PC — PC AT — было выпущено в 1984 г. **AT** означало **Advanced Technology** (улучшенная технология) и указывало на некото-

рые существенные улучшения по сравнению с оригинальным ПК, которые в основном были результатом использования процессора Intel 80286.

Как и 8088, 80286 был 16-разрядным процессором, который поддерживал обратную совместимость с 8088: код 8088 можно было выполнять на 80286 без изменений. 80286 имел 16-разрядную шину передачи данных и 24 адресные строки, поддерживающие адресное пространство размером 16 Мбайт. 16-разрядная внешняя шина данных повышала скорость доступа к данным по сравнению с 8-разрядной шиной процессора 8088. Скорость выполнения инструкций (за такт процессора) была примерно в два раза выше, чем у 8088, во многих приложениях. Это означало, что при той же тактовой частоте 80286 мог работать в два раза быстрее, чем 8088. Процессор оригинального PC AT имел тактовую частоту 6 МГц, а его более поздняя версия работала на частоте 8 МГц. Скорость выполнения инструкций на 80286 с тактовой частотой 6 МГц достигала примерно 0,9 млн инструкций в секунду, что было примерно в три раза выше, чем у 8088.

В 80286 был реализован режим защищенной виртуальной адресации, предназначенный для поддержки многопользовательских операционных систем и многозадачности.

В защищенном режиме процессор обеспечивал защиту памяти, чтобы гарантировать, что программы одного пользователя не окажут влияния на работу операционной системы или программ других пользователей. Это революционное технологическое достижение в области персональных компьютеров оставалось мало используемым в течение многих лет главным образом из-за непомерно высокой стоимости добавления в компьютерную систему достаточного объема памяти, чтобы сделать ее полезной в многопользовательском, многозадачном контексте.

После 80286 следующее поколение линейки процессоров *x*86 было представлено моделью 80386, выпущенной в 1985 г. 80386 был 32-разрядным процессором с поддержкой 32-разрядной линейной модели памяти в защищенном режиме. Линейная модель позволяла программистам напрямую обращаться к 4 Гбайт памяти, не требуя манипуляций с сегментными регистрами. В 1986 г. компания Compaq представила совместимый с IBM PC персональный компьютер на базе 80386, названный DeskPro. DeskPro поставляется с версией ОС Microsoft Windows, ориентированной на архитектуру 80386.

Процессор 80386 в значительной степени поддерживал обратную совместимость с процессорами 80286 и 8088. Архитектура, реализованная в 80386, остается текущей стандартной архитектурой семейства *x*86. Мы подробно обсудим ее в *главе 10*.

Начальная версия 80386 имела тактовую частоту 33 МГц, а скорость выполнения достигала примерно 11,4 млн инструкций в секунду. Современные реализации архитектуры *x*86 работают в несколько сотен раз быстрее, чем исходная версия, за счет более высокой тактовой частоты, повышения производительности, включая широкое использование многоуровневой кеш-памяти, и более эффективного выполнения инструкций на аппаратном уровне. Мы рассмотрим преимущества кеш-памяти в *главе 8*.

iPhone

В 2007 г. Стив Джобс представил миру iPhone — человечество тогда и понятия не имело о том, какую пользу можно было извлечь из такого устройства. iPhone был создан на основе предыдущих революционных достижений Apple Computer, включая компьютер Macintosh, выпущенный в 1984 г., и музыкальный плеер iPod, появившийся в 2001 г. iPhone сочетал в себе функции iPod, мобильного телефона и компьютера с выходом в Интернет.

В iPhone была упразднена аппаратная клавиатура, которая была распространена на смартфонах того времени, ее заменил сенсорный экран, способный отображать экранную клавиатуру или любой другой тип пользовательского интерфейса. Помимо прикосновений для выбора символов на клавиатуре и нажатия кнопок экран поддерживал жесты несколькими пальцами, которые позволяли выполнять такие действия, как масштабирование фотографий.

iPhone работал под управлением операционной системы OS X; та же ОС использовалась на флагманских компьютерах Macintosh того времени.

Такое решение сразу же позволило iPhone поддерживать широкий спектр приложений, уже разработанных для Mac, а разработчики ПО получили возможность быстро внедрять новые приложения, адаптированные для iPhone, после того, как Apple разрешила стороннюю разработку приложений.

iPhone 1 был оснащен 3,5-дюймовым экраном с разрешением 320×480 пикселов. Он имел толщину 11,6 мм (тоньше, чем у других смартфонов), встроенную 2-мегапиксельную камеру и весил 136 г. Когда пользователь подносил телефон к уху, датчик приближения обнаруживал это и отключал подсветку экрана и восприятие касаний экрана на время разговора. В устройстве также был предусмотрен датчик внешней освещенности для автоматической настройки яркости экрана и акселерометр, который определял выбор портретной или альбомной ориентации экрана.

iPhone 1 содержал 128 Мбайт оперативной памяти и 4, 8 или 16 Гбайт флеш-памяти и поддерживал стандарт **глобальной системы мобильной связи** (Global System for Mobile communications, GSM), Wi-Fi (802.11b/g) и Bluetooth.

В отличие от обилия общедоступной информации об IBM PC, компания Apple, как правило, не раскрывала детали архитектуры iPhone. Apple не опубликовала никакой информации о процессоре или других внутренних компонентах первого iPhone, просто назвав его **закрытой системой**.

Несмотря на отсутствие официальной информации от Apple, нашлись энтузиасты, которые разобрали различные модели iPhone и попытались идентифицировать компоненты телефона и взаимосвязи между ними. Исследователи программного обеспечения разработали различные тесты, которые пытались определить конкретную модель процессора и других цифровых устройств, установленных в iPhone. Такие усилия по обратной разработке подвержены ошибкам, поэтому описания архитектуры iPhone в этом разделе следует воспринимать с определенной долей скептицизма.

В iPhone 1 был установлен выпускаемый компанией Samsung 32-разрядный процессор ARM11 с тактовой частотой 412 МГц. ARM11 был улучшенным вариантом процессоров ARM предыдущего поколения и включал 8-ступенчатый конвейер обработки инструкций и поддержку модели обработки с **одним потоком инструкций и множеством потоков данных** (single instruction, multiple data, SIMD) для улучшения качества звука и видео. Архитектуру процессора ARM мы подробно обсудим в *главе 10*.

Питание iPhone 1 обеспечивал литий-ионный полимерный аккумулятор напряжением 3,7 В. Замена аккумулятора не предусматривалась, и, по оценкам Apple, после 400 циклов зарядки и разрядки он должен был потерять около 20% своей первоначальной емкости. Apple заявила о том, что устройство будет работать до 250 часов в режиме ожидания и до 8 часов в режиме разговора без подзарядки.

Через 6 месяцев после появления iPhone журнал "Time" назвал iPhone "Изобретением года" за 2007 г. В 2017 г. "Time" составил рейтинг *50 самых влиятельных устройств всех времен*. iPhone возглавил этот список.

В следующем разделе мы рассмотрим взаимосвязь развития технологических достижений в области вычислительной техники и лежащих в их основе физических ограничений интегральных схем на базе кремния.

Закон Мура

Для тех, кто работает в быстро развивающейся области компьютерных технологий, составление планов на будущее является сложной задачей. Это справедливо независимо от того, ставите ли вы перед собой цель спланировать собственную карьеру или разработать оптимальный график инвестиций в исследования и разработки для крупной полупроводниковой корпорации. Никто и никогда не может быть полностью уверен, каким будет следующий технологический скачок, какие из его последствий повлияют на отрасль и пользователей или когда он произойдет. Один из подходов, который оказался полезным в этих сложных условиях, заключается в разработке эмпирического правила или закона, основанного на опыте.

В 1957 г. Гордон Мур стал соучредителем компании Fairchild Semiconductor, а позже занимал посты председателя Совета директоров и генерального директора Intel. В 1965 г. Мур опубликовал в журнале "Electronics" статью, где он предложил свой прогноз изменений, которые произойдут в полупроводниковой промышленности в течение следующих 10 лет. В статье он отметил, что количество дискретных компонентов, таких как транзисторы, диоды и конденсаторы, которые можно было разместить на одном чипе, удваивалось примерно каждый год, и что эта тенденция, вероятно, сохранится в течение следующих 10 лет. Эта формула удвоения стала известна как **закон Мура**. Это не был научный закон наподобие закона всемирного тяготения. Скорее это был результат наблюдения за историческими тенденциями, и Мур посчитал, что такая формулировка может определенным образом прогнозировать будущее.

Для тех десяти лет закон Мура оказался впечатляюще точным. В 1975 г. он пересмотрел прогнозируемый темп роста на следующее десятилетие, предположив, что количество компонентов на интегральной схеме будет удваиваться каждые два года, а не ежегодно. Этот темп выдерживался десятилетиями — примерно до 2010 г. В последние годы темпы роста несколько снизились. В 2015 г. Брайан Кржанич, генеральный директор корпорации Intel, заявил, что темпы роста компании замедляются и удвоение достигается примерно каждые два с половиной года.

Несмотря на то что срок удвоения плотности интегральных схем увеличивается, нынешние темпы роста представляют собой феноменальный прогресс, который, как можно ожидать, продолжится и в будущем, но не так быстро, как раньше.

Закон Мура на протяжении десятилетий зарекомендовал себя как надежный инструмент для оценки деятельности полупроводниковых компаний.

Компании использовали его для постановки целей по производительности своих продуктов и для планирования своих инвестиций. Сравнивая увеличение плотности интегральных схем для продуктов компании с предыдущими значениями и с показателями других компаний, руководители полупроводниковых компаний и отраслевые аналитики могут оценивать и квалифицировать эффективность компании. Результаты такого анализа напрямую повлияли на решения об инвестировании в огромные новые заводы по производству интегральных схем и достижении новых уровней интеграции за счет уменьшения размеров элементов.

За десятилетия, прошедшие с момента появления IBM PC, возможности однокристальных микропроцессоров значительно расширились. Процессоры современных поколений в сотни раз быстрее, изначально работают с 32- и 64-битными данными, имеют гораздо больше встроенной памяти и раскрывают намного больше функциональных возможностей, и все это — на одной интегральной схеме.

Эти усовершенствования стали возможными благодаря увеличению плотности размещения полупроводниковых элементов, как предсказывает закон Мура. Транзисторы меньшего размера работают на более высоких тактовых частотах из-за более коротких соединений между элементами схемы. Кроме того, меньшие транзисторы, очевидно, позволяют реализовать больше функций на заданной площади кристалла. Из-за меньших размеров и расстояний между соседними компонентами транзисторы потребляют меньше энергии и выделяют меньше тепла.

В законе Мура не было ничего магического. Это было лишь наблюдение за тенденциями того времени. Одной из тенденций был неуклонный рост размеров полупроводниковых кристаллов. Это стало результатом улучшения производственных процессов, которые снизили плотность дефектов, что позволило получить приемлемый выход продукции с более крупными кристаллами интегральных схем. Другой тенденцией было постоянное уменьшение размера самых маленьких компонентов, которые можно было надежно изготовить в составе схемы. Последняя тенденция была обусловлена тем, что Мур называл "изобретательностью" разработчиков схем, которая выражалась во все более эффективном и действенном использовании растущего числа схемных элементов, размещаемых на кристалле.

Традиционные процессы производства полупроводников начали приближаться к физическим пределам, которые в конечном счете затормозят рост, предсказанный законом Мура. Наименьшие элементы современных коммерчески доступных интегральных схем имеют размер около 5 **нанометров (нм)**. Для сравнения: типичный человеческий волос имеет толщину около 50 000 нм, а молекула воды (одна из самых маленьких молекул) имеет размер 0,28 нм. Существует предел, ограничивающий дальнейшее уменьшение схемных элементов, т. к. их размеры приближаются к атомарному масштабу.

В дополнение к проблеме создания надежных элементов из небольшого количества молекул начинают проявляться и другие физические эффекты, такие как *дифракционный предел Аббе*, которые могут стать серьезным препятствием для производства одноразрядных схем нанометрового масштаба.

Мы не будем вдаваться в подробности этих явлений, достаточно знать, что неуклонное увеличение плотности компонентов интегральных схем, которое продолжалось в течение десятилетий в соответствии с законом Мура, в ближайшие годы станет намного труднее поддерживать.

Это не означает, что мы "застрянем" на уровне процессоров, которые будут во многом аналогичны тем, что сейчас доступны на рынке. Несмотря на замедление роста плотности транзисторов, производители полупроводников используют ряд альтернативных методов для дальнейшего увеличения мощности вычислительных устройств. Одним из подходов является специализация, при которой схемы разрабатываются для высокоэффективного решения задач определенной категории, а не просто для удовлетворительного выполнения широкого круга задач.

Графические процессоры (graphics processing units, GPU) являются превосходным примером специализации. Графические процессоры первого поколения были нацелены исключительно на повышение скорости построения трехмерных графических сцен, в основном для использования в видеоиграх. Расчеты, необходимые для построения трехмерной сцены, хорошо определены и должны применяться к тысячам пикселов для создания одного кадра. Этот процесс повторяется для каждого последующего кадра, а чтобы обеспечить удовлетворительное взаимодействие с пользователем, кадры должны перерисовываться с частотой 60 Гц или выше. Требовательный к вычислительным ресурсам и циклический характер этой задачи идеально подходит для ускорения с помощью аппаратного распараллеливания. Несколько вычислительных модулей внутри графического процессора одновременно выполняют практически одни и те же вычисления с разными входными данными для получения отдельных потоков выходных данных. Эти выходные потоки объединяются для создания полной сцены. Архитектура современных графических процессоров была усовершенствована для поддержки других областей применения вычислений, таких как обучение нейронных сетей на больших объемах данных. Архитектуры графических процессоров будут подробно описаны в *главе 6*.

В ближайшие годы тенденция, описанная законом Мура, продемонстрирует признаки ослабления. Так какие достижения могут компенсировать этот спад, чтобы запустить следующий раунд инноваций в компьютерных архитектурах? Сегодня

мы не знаем наверняка, но некоторые привлекательные варианты в настоящее время пристально изучаются. Квантовые вычисления — один из примеров таких технологий. Мы рассмотрим эту технологию в *главе 17*.

Квантовые вычисления используют свойства субатомных частиц для обработки данных таким образом, который недоступен для традиционных компьютеров. Основным элементом квантовых вычислений является **кубит**, или квантовый бит. Кубит подобен обычному двоичному биту, но помимо представления состояний 0 и 1, кубиты могут достигать состояния, являющегося суперпозицией (или смесью) состояний 0 и 1. При измерении выходное значение кубита всегда будет равно 0 или 1, но вероятность получения любого из выходных значений зависит от квантового состояния кубита перед считыванием. Для того чтобы воспользоваться уникальными возможностями квантовых вычислений, необходимы специализированные алгоритмы.

Другая перспективная возможность состоит в том, что следующий значительный технологический прорыв в вычислительных устройствах будет связан с чем-то, о чем мы либо не думали, либо, если мы думали об этом, то могли сразу отбросить эту идею как нереалистичную. iPhone, обсуждавшийся в предыдущем разделе, является примером инновационного продукта, создавшего новую категорию, — это устройство произвело революцию в личном общении и открыло новые способы использования Интернета. Следующим крупным достижением может быть новый тип продукта, удивительная новая технология или некое сочетание продукта и технологии. Прямо сейчас мы не знаем, что это будет и когда это произойдет, но мы с уверенностью можем сказать, что такие изменения грядут.

В следующем разделе представлены некоторые фундаментальные концепции цифровых вычислений, которые необходимо понять, прежде чем мы углубимся в цифровые схемы и детали архитектуры современных компьютеров в следующих главах.

Архитектура компьютеров

Описания ряда ключевых архитектур из истории вычислительной техники, представленные в предыдущих разделах этой главы, содержали некоторые знакомые или незнакомые вам термины. В этом разделе будут представлены концептуальные структурные элементы, которые используются для создания современных процессоров и связанных с ними компьютерных подсистем.

Представление чисел уровнями напряжения

Одной из общепринятых особенностей современных компьютеров является использование уровней напряжения для указания значений данных. Обычно распознаются только два уровня напряжения: низкий и высокий. Низкому уровню часто присваивают значение 0, а высокому уровню — значение 1.

Напряжение в любой точке цепи (цифровой или иной) имеет аналоговую природу и может принимать любое значение в пределах своего рабочего диапазона. При пере-

ходе с низкого уровня на высокий или обратно напряжение должно проходить через все промежуточные уровни. В контексте цифровых схем переходы между низкими и высокими уровнями происходят быстро, и эти схемы проектируются таким образом, чтобы не реагировать на напряжения между высоким и низким уровнями.

Двоичные и шестнадцатеричные числа

Схемы внутри процессора ни в каком из возможных смыслов не работают напрямую с числами. Схемные элементы процессора подчиняются законам электричества и электроники и просто реагируют на поступающие входные сигналы. Входные сигналы, которые управляют этими действиями, порождаются кодом, разработанным программистами, и данными, предоставленными в качестве входных данных для программы. Интерпретация результатов работы программы, например чисел в электронной таблице или символов в программе обработки текстов, — это чисто человеческая интерпретация, придающая значение результату электронных взаимодействий внутри процессора. Решение присвоить 0 низкому напряжению и 1 высокому напряжению является первым шагом в процессе этой интерпретации.

Наименьшей единицей информации в цифровом компьютере является двоичная цифра, называемая **битом**, которая представляет собой отдельный элемент данных, содержащий значение 0 или 1. Несколько битов можно сгруппировать, чтобы обеспечить представление большего диапазона значений. **Байт** состоит из 8 битов, объединенных в одно значение. Байт — это наименьшая единица информации, которая может быть прочитана или записана в память большинством современных процессоров. Некоторые компьютеры прошлого и настоящего используют разное количество битов для наименьшего адресуемого элемента данных, но 8-битный байт представляет наиболее распространенный размер.

Один бит может принимать два значения: 0 или 1. Два сгруппированных бита могут принимать четыре значения: 00, 01, 10 и 11. Три бита могут принимать восемь значений: 000, 001, 010, 011, 100, 101, 110 и 111. В целом группа из *n* битов может принимать 2^n значений. Таким образом, 8-битный байт может представлять 2^8 или 256 уникальных значений.

Когда дело доходит до выполнения арифметических операций, двоичный формат представления чисел не является предпочтительным для большинства людей. Работа с числами, представленными в виде 11101010, может быть достаточно сложной и подверженной ошибкам, особенно если речь идет о манипулировании 32- и 64-битными значениями. Для того чтобы упростить работу с этими числами, часто вместо них используют **шестнадцатеричные числа**. Термин "шестнадцатеричный" (hexadecimal) часто сокращают до *hex*.

В шестнадцатеричной системе счисления двоичные числа разделены на группы по четыре бита. При 4 битах в группе количество возможных значений равно 2^4 или 16. Первые десять из этих 16 чисел обозначаются цифрами от 0 до 9, а оставшиеся шесть — буквами от A до F. В табл. 1.1 показаны первые 16 двоичных значений,

начиная с 0, вместе с соответствующей шестнадцатеричной цифрой и десятичным эквивалентом двоичного и шестнадцатеричного значений.

Таблица 1.1. *Двоичные, шестнадцатеричные и десятичные числа*

Двоичное	Шестнадцатеричное	Десятичное
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Двоичное число 11101010 можно представить более компактно, разбив его на две группы по четыре бита (1110 и 1010) и записав их в виде шестнадцатеричных цифр EA. Группу из четырех битов иногда называют *полубайтом*, т. е. половиной байта. Двоичные числа могут принимать только два значения, поэтому двоичная система является системой счисления с основанием 2. Шестнадцатеричные числа могут принимать 16 значений, поэтому шестнадцатеричная система счисления имеет основание 16. Десятичные числа могут иметь 10 значений, поэтому десятичная система имеет основание 10.

При работе с различными системами счисления легко запутаться. Является ли число, записанное как 100, двоичным, шестнадцатеричным или десятичным? Без дополнительной информации ответить на этот вопрос невозможно. Различные языки программирования и учебники использовали разные подходы к устранению подобной двусмысленности. В большинстве случаев десятичные числа не имеют дополнительных обозначений, поэтому число, записанное как 100, обычно является деся-

тичным. В языках программирования, таких как C и C++, перед шестнадцатеричными числами ставится обозначение `0x`, поэтому число в виде `0x100` представляет собой 100 в шестнадцатеричном формате. В языках ассемблера для обозначения шестнадцатеричных чисел могут использоваться либо префикс `$`, либо суффикс `h`. Использование двоичных значений в программировании менее распространено — в основном это обусловлено тем, что шестнадцатеричные значения предпочтительнее из-за их компактности. Некоторые компиляторы поддерживают использование `0b` в качестве префикса для двоичных чисел.

ПРЕДСТАВЛЕНИЕ ШЕСТНАДЦАТЕРИЧНОГО ЧИСЛА



В этой книге для представления шестнадцатеричных чисел, в зависимости от контекста, используется либо префикс `$`, либо суффикс `h`. Суффикс `b` обозначает двоичные числа, а отсутствие префикса или суффикса указывает на десятичные числа.

Биты в двоичном числе нумеруются индивидуально, причем бит 0 в крайней правой позиции является наименее значащим битом. Номера битов увеличиваются по величине справа налево, вплоть до самого старшего бита в крайней левой позиции.

Некоторые примеры помогут прояснить эти правила. В табл. 1.1 в двоичном значении 0001b (1 в десятичном формате) для бита с номером 0 установлена 1, а остальные три бита равны 0. Для 0010b (2 в десятичном формате) бит 1 установлен, а остальные биты обнулены. Для 0100b (4 в десятичном формате) бит 2 установлен, а остальные биты обнулены.

УСТАНОВЛЕННЫЙ БИТ И ОБНУЛЕННЫЙ БИТ



Установленный бит имеет значение 1. Обнуленный бит имеет значение 0.

8-битный байт может принимать значения от $00h до $FF, эквивалент десятичного диапазона 0–255. При выполнении сложения на уровне байтов результат может превышать 8 битов. Например, добавление $01 к $FF дает результат $100. При использовании 8-битных регистров это означает операцию переноса в девятый бит, которая должна соответствующим образом обрабатываться аппаратными средствами процессора и программным обеспечением, выполняющим сложение.

В беззнаковой арифметике вычитание $01 из $00 дает значение $FF, т. е. выполняется циклический переход к значению $FF. В зависимости от выполняемых вычислений это может быть желаемым результатом, а может и не быть им. Опять же, аппаратные средства и программное обеспечение процессора должны обработать эту ситуацию, чтобы прийти к желаемому результату.

При необходимости отрицательные значения могут быть представлены с помощью двоичных чисел. Наиболее распространенным форматом чисел со знаком в современных процессорах является **дополнительный код**. При его использовании 8-битные числа со знаком охватывают диапазон от –128 до 127. Старший бит значения в дополнительном коде является битом знака: 0 в этом бите представляет положительное число, а 1 — отрицательное число. Отрицание числа в дополнительном коде (умножение на –1) выполняется путем инвертирования всех его битов, добавления 1 и игнорирования переноса. Инверсия бита означает изменение состояния обнуленного бита на 1, а установленного бита — на 0. Ряд пошаговых примеров отрицания 8-битных чисел со знаком приведен в табл. 1.2.

Таблица 1.2. *Примеры операции отрицания*

Десятичное значение	Двоичное значение	Инверсия битов	Добавление единицы	Результат отрицания
0	00000000b	11111111b	00000000b	0
1	00000001b	11111110b	11111111b	–1
–1	11111111b	00000000b	00000001b	1
127	01111111b	10000000b	10000001b	–127
–127	10000001b	01111110b	01111111b	127

Отрицание 0 возвращает 0, как и следовало ожидать с точки зрения математики.



АРИФМЕТИКА ДОПОЛНИТЕЛЬНОГО КОДА

На уровне битов арифметика дополнительного кода идентична беззнаковой арифметике. Манипуляции, связанные со сложением и вычитанием, одинаковы независимо от того, являются ли входные значения знаковыми или беззнаковыми. Интерпретация результата как знакового или беззнакового полностью зависит от намерения пользователя.

В табл. 1.3 показано соответствие двоичных значений от 00000000b до 11111111b значениям со знаком в диапазоне от –128 до 127 и беззнаковым значениям от 0 до 255.

Таблица 1.3. *Знаковые и беззнаковые 8-битные числа*

Двоичное	Десятичное со знаком	Десятичное без знака
00000000b	0	0
00000001b	1	1
00000010b	2	2
...	...	...

Таблица 1.3 (окончание)

Двоичное	Десятичное со знаком	Десятичное без знака
01111110b	126	126
01111111b	127	127
10000000b	–128	128
10000001b	–127	129
10000010b	–126	130
...	...	...
11111101b	–3	253
11111110b	–2	254
11111111b	–1	255

Знаковые и беззнаковые представления двоичных чисел распространяются на более крупные целочисленные типы данных. 16-битные значения могут представлять целые числа без знака от 0 до 65 535 и целые числа со знаком в диапазоне от –32 768 до 32 767. В современных процессорах и языках программирования обычно доступны 32-битные, 64-битные и даже более крупные целочисленные типы данных.

Микропроцессор 6502

В этом разделе представлена процессорная архитектура, являющаяся относительно простой по сравнению с более мощными современными процессорами.

Цель такого подхода — представление некоторых базовых концепций, общих для широкого круга процессоров, от бюджетных микроконтроллеров до сложных многоядерных 64-разрядных процессоров.

Процессор 6502 был представлен компанией MOS Technology в 1975 г. Он нашел широкое применение в игровых консолях от Atari и Nintendo, а также в компьютерах, продаваемых Commodore и Apple. Различные версии процессора 6502 по-прежнему широко используются во встроенных системах и сегодня — согласно некоторым оценкам, с 2018 г. их выпущено от 5 до 10 млрд (да-да, миллиардов). В поп-культуре, и робот Бендер из "Футурамы", и Т-800 из "Терминатора", судя по ряду кадров, работали на 6502.

Как и многие ранние микропроцессоры, 6502 использовал источник питания напряжением **5 вольт (В) постоянного тока**. В таких схемах низкий уровень сигнала — это любое напряжение от 0 до 0,8 В, а высокий уровень — любое напряжение от 2 до 5 В. Напряжения между этими диапазонами возникают только при переходах от низкого к высокому и от высокого к низкому уровню. Низкий уровень сигнала определяется как логический 0, а высокий уровень сигнала — как логическая 1.

В *главе 2* будут подробно рассмотрены электронные схемы, используемые в цифровой электронике.

Длина слова процессора определяет размер основного элемента данных, с которым работает процессор. Длина слова процессора 6502 равна 8 битам. Это означает, что 6502 считывает и записывает данные в память по 8 бит за раз и хранит данные в 8-разрядных регистрах.

Для памяти программ и памяти данных используется одно и то же адресное пространство, а доступ к памяти 6502 осуществляется по одной шине. Как и в Intel 8088, в 6502 реализована архитектура фон Неймана. Этот процессор имеет 16-разрядную адресную шину, позволяющую адресовать 64 Кбайт памяти.

1 Кбайт содержит 2^{10} или 1024 байта. Количество уникальных двоичных комбинаций для 16 адресных строк равно 2^{16}, что позволяет получить доступ к 65 536 ячейкам памяти с байтовой организацией. Обратите внимание, что способность устройства адресовать 64 Кбайт не означает, что каждой из этих ячеек должна соответствовать физическая память. Компьютер Commodore VIC-20, основанный на процессоре 6502, содержал всего 5 Кбайт ОЗУ и 20 Кбайт ПЗУ.

Процессор 6502 содержит внутренние области хранения данных, называемые регистрами. **Регистр** — это место в логическом устройстве, в котором слово информации может храниться и обрабатываться во время вычислений. Типичный процессор содержит небольшое количество регистров для временного хранения значений данных и выполнения таких операций, как сложение или вычисления адреса.

На рис. 1.1 показана структура регистров 6502. Процессор содержит пять 8-разрядных регистров (A, X, Y, P и S) и один 16-разрядный регистр (PC). Цифры над каждым регистром указывают номера битов на каждом из концов регистра.

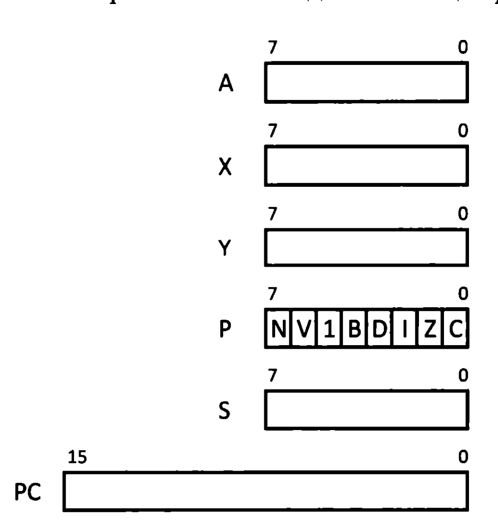


Рис. 1.1. *Набор регистров процессора 6502*

Каждый из регистров A, X и Y может служить хранилищем общего назначения. Инструкции программы могут загружать значение в один из этих регистров и после

выполнения нескольких других инструкций использовать сохраненное значение для какой-либо цели, если промежуточные инструкции не изменили содержимое этого регистра. Регистр A — это единственный регистр, поддерживающий выполнение арифметических операций. Регистры X и Y, но не регистр A, можно использовать в качестве индексных регистров при вычислении адресов памяти.

Регистр P содержит флаги процессора. Каждый бит в этом регистре имеет уникальное назначение, за исключением бита, помеченного 1. Бит 1 не используется и может быть проигнорирован. Остальные биты в этом регистре называются **флагами** и указывают на определенное состояние или представляют собой параметр конфигурации. В процессоре 6502 используются следующие флаги.

- N — флаг отрицательного знака. Этот флаг устанавливается, когда в результате арифметической операции устанавливается бит 7. Используется в знаковой арифметике.
- V — флаг переполнения. Он устанавливается, когда операция сложения или вычитания над числами со знаком приводит к положительному или отрицательному переполнению — выходу за пределы диапазона от –128 до 127.
- B — флаг программного прерывания. Он указывает на выполнение инструкции Break (BRK). Этот бит отсутствует в самом регистре P. Значение флага B учитывается только при проверке содержимого регистра P, сохраненного в стеке с помощью инструкции BRK или прерывания. Флаг B устанавливается для того, чтобы при обработке прерываний отличать программное прерывание, возникшее в результате выполнения инструкции BRK, от аппаратного прерывания.
- D — флаг десятичного режима. Установка этого флага означает, что арифметика процессора будет работать в двоично-десятичном (binary-coded decimal, BCD) режиме. Режим BCD используется редко и здесь обсуждаться не будет. Следует отметить, что этот режим вычислений с основанием 10 напоминает архитектуры аналитической машины Бэббиджа и ENIAC.
- I — флаг запрета прерываний. Его установка указывает, что входы прерываний (кроме немаскируемого прерывания) обрабатываться не будут.
- Z — флаг нуля. Он устанавливается, когда операция приводит к результату, равному 0.
- C — флаг переноса. Он устанавливается, когда арифметическая операция приводит к переносу.

Флаги N, V, Z и C являются наиболее важными флагами в контексте общих вычислений, включающих циклы, подсчет и арифметику.

Регистр S является **указателем стека**. В процессоре 6502 стек — это область памяти, расположенная по адресам от $100 до $1FF. Эта область длиной 256 байтов используется для временного хранения параметров подпрограмм и содержит обратный адрес при вызове подпрограммы. При запуске системы регистр S инициализируется таким образом, чтобы указывать на верхнюю часть этой области. Значения

"проталкиваются" в стек с помощью таких инструкций, как PHA, которая помещает в стек содержимое регистра A.

Когда значение помещается в стек, процессор 6502 сохраняет это значение по адресу, указанному регистром S, после добавления фиксированного смещения $100, а затем уменьшает значение регистра S на единицу. Дополнительные значения могут быть помещены в стек путем выполнения дополнительных инструкций записи в стек. По мере ввода дополнительных значений стек в памяти растет в направлении вниз. Программы должны следить за тем, чтобы не превышать зафиксированный для стека размер в 256 байт при загрузке в него данных.

Хранящиеся в стеке данные должны извлекаться в порядке, обратном тому, в котором они были помещены в стек. Стек представляет собой структуру данных, организованную по принципу **"последним вошел, первым вышел"** (last in, first out, LIFO). Это означает, что извлекаемое из стека значение — это байт, который был добавлен в него последним. Инструкция PLA увеличивает значение регистра S на 1, затем копирует значение по адресу, указанному регистром S (плюс смещение $100), в регистр A.

Регистр PC — это счетчик инструкций. Он содержит адрес инструкции, которая должна быть выполнена следующей. В отличие от других регистров, PC имеет размер 16 бит, что обеспечивает доступ ко всему адресному пространству процессора 6502.

Каждая инструкция состоит из 1-байтового кода операции, для краткости называемого **опкодом** (opcode), и может сопровождаться байтами операндов (от 0 до 2), в зависимости от типа инструкции. После выполнения каждой инструкции содержимое регистра PC обновляется, чтобы указать на инструкцию, следующую за только что выполненной. В дополнение к автоматическим обновлениям во время последовательного выполнения инструкций регистр PC может быть изменен инструкциями перехода, ветвления, вызова подпрограмм и возврата из них.

Набор инструкций микропроцессора 6502

Теперь рассмотрим набор инструкций процессора 6502. Инструкции — это отдельные команды процессора, которые выстраиваются в последовательность для выполнения алгоритма, закодированного программистом. Инструкция содержит двоичное число, называемое кодом операции (или **опкодом**), которое сообщает процессору, что следует делать при выполнении этой инструкции.

При желании программисты могут писать код непосредственно с помощью инструкций процессора. Примеры этого мы увидим позже в этом разделе. Программисты также могут создавать код на так называемом языке высокого уровня. Затем программист использует программный инструмент, называемый **компилятором**, который преобразует высокоуровневый код в последовательность инструкций процессора (обычно гораздо более длинную).

В этом разделе мы работаем с кодом, написанным в виде последовательностей инструкций процессора. Эта форма исходного кода называется **языком ассемблера**.

Каждая инструкция процессора 6502 имеет трехсимвольное мнемоническое обозначение. В исходных файлах языка ассемблера каждая строка кода содержит мнемоническое обозначение инструкции, за которым следуют любые операнды, относящиеся к этой инструкции. Комбинация мнемонического обозначения и операндов определяет **режим адресации**. Процессор 6502 поддерживает несколько режимов адресации, обеспечивая гибкость доступа к данным в регистрах и памяти. Для этого введения мы будем работать только с режимом **непосредственной** адресации, в котором сам операнд содержит значение, а не указывает регистр или ячейку памяти, содержащую нужное значение. Непосредственному значению предшествует символ `#`.

В ассемблере процессора 6502 десятичные числа не имеют дополнительных обозначений (48 означает число 48 в десятичном формате), в то время как шестнадцатеричным значениям предшествует символ `$` ($30 означает число 30 в шестнадцатеричном формате, эквивалент 00110000b и десятичного числа 48). Непосредственное десятичное значение выглядит как #48, а шестнадцатеричное значение — как #$30.

Некоторые примеры ассемблерного кода продемонстрируют арифметические возможности процессора 6502. В следующих примерах используются пять инструкций 6502:

- `LDA` загружает значение в регистр A;
- `ADC` выполняет сложение с учетом переноса (флаг C в регистре P) в качестве дополнительных входных и выходных данных;
- `SBC` выполняет вычитание с учетом флага C в качестве дополнительных входных и выходных данных;
- `SEC` устанавливает флаг C непосредственно;
- `CLC` обнуляет флаг C непосредственно.

Так как флаг C предоставляет входные данные для инструкций сложения и вычитания, перед выполнением инструкции `ADC` или `SBC` важно убедиться, что он имеет правильное значение. Перед выполнением операции сложения флаг C должен быть обнулен, чтобы указать на отсутствие переноса из предыдущей операции сложения. При сложении многобайтовых значений (например, при работе с 16-, 32- или 64-битными числами) перенос, если таковой имеется, будет распространяться от суммы одной пары байтов к следующей по мере добавления более значимых байтов. Если флаг C установлен при выполнении инструкции `ADC`, то к результату следует добавить 1. После выполнения инструкции `ADC` флаг C выступает в качестве девятого бита результата: если флаг C имеет значение 0, переноса не было, а 1 указывает на то, что из 8-разрядного регистра был выполнен перенос.

Вычитание с использованием инструкции `SBC` может немного сбить с толку начинающих программистов на языке ассемблера процессора 6502. Школьники, изучающие вычитание, используют технику заимствования при вычитании большей цифры из меньшей. В процессоре 6502 флаг C представляет собой признак, противоположный заимствованию. Если C имеет значение 1, то заем равен 0, а если

С равно 0, то заем равен 1. Выполнение простого вычитания без входящего заимствования требует установки флага С перед выполнением инструкции SBC.

В примерах из табл. 1.4 процессор 6502 используется в качестве калькулятора с входными данными, определяемыми в виде непосредственных значений в коде, и с результатом, сохраняемым в регистре А. Столбец "Результаты" содержит окончательное значение регистра А и состояния флагов N, V, Z и С.

Таблица 1.4. *Последовательности арифметических инструкций процессора 6502*

Последовательность инструкций	Описание	Результаты				
		A	N	V	Z	C
CLC LDA #1 ADC #1	8-битное сложение без переноса: обнуление флага переноса, затем загрузка непосредственного значения 1 в регистр А и добавление к нему 1	$02	0	0	0	0
SEC LDA #1 ADC #1	8-битное сложение с переносом: установка флага переноса, затем загрузка непосредственного значения 1 в регистр А и добавление к нему 1	$03	0	0	0	0
SEC LDA #1 SBC #1	8-битное вычитание без заимствования: установка флага переноса, затем загрузка непосредственного значения 1 в регистр А и вычитание из него 1. С = 1 означает отсутствие заимствования	$00	0	0	1	1
CLC LDA #1 SBC #1	8-битное вычитание с заимствованием: обнуление флага переноса, затем загрузка непосредственного значения 1 в регистр А и вычитание из него 1. С = 0 указывает на наличие заимствования	$FF	1	0	0	0
CLC LDA $FF ADC #1	Положительное переполнение без знака: добавление 1 к $FF. С = 1 указывает на наличие переноса	$00	0	0	1	1
SEC LDA #0 SBC #1	Отрицательное переполнение без знака: вычитание 1 из 0. С = 0 указывает на наличие заимствования	$FF	1	0	0	0
CLC LDA #$7F ADC #1	Положительное переполнение со знаком: добавление 1 к $7F. С = 1 указывает на наличие положительного переполнения со знаком	$80	1	1	0	0

Таблица 1.4 (окончание)

Последовательность инструкций	Описание	Результаты				
		A	N	V	Z	C
SEC LDA #$80 SBC #1	Отрицательное переполнение со знаком: вычитание 1 из $80. C = 1 указывает на наличие отрицательного переполнения со знаком	$7F	0	1	0	1

Если у вас под рукой нет компьютера на базе процессора 6502 с ассемблером и отладчиком, в Интернете доступно несколько его бесплатных эмуляторов, которые вы можете запустить в своем веб-браузере. Один превосходный эмулятор доступен по адресу **https://skilldrick.github.io/easy6502/**. Зайдите на этот веб-сайт и прокрутите содержимое окна вниз, пока не найдете листинг кода по умолчанию с кнопками для сборки и запуска кода 6502. Замените листинг кода по умолчанию группой из трех инструкций из табл. 1.4, затем выполните код.

Для того чтобы изучить действие каждой инструкции в последовательности, используйте элементы управления отладчика для пошагового выполнения инструкций и наблюдения за результатом выполнения каждой инструкции в регистрах процессора.

В этом разделе представлено очень краткое введение в процессор 6502 и рассмотрена лишь небольшая часть его возможностей. Одна из целей этого рассмотрения состояла в том, чтобы проиллюстрировать сложность решения проблемы переносов при выполнении сложения и заимствований при выполнении вычитания. Начиная с Чарльза Бэббиджа и заканчивая проектировщиками процессора 6502 и разработчиками современных вычислительных систем, архитекторы компьютеров находили решения задач вычислений и воплощали их в жизнь с применением наилучших доступных им технологий.

Резюме

Эта глава началась с краткой истории автоматических вычислительных устройств, за которой последовал обзор значительных технологических достижений, приведших к ряду прорывов в вычислительных возможностях. Затем последовало обсуждение закона Мура с оценкой его применимости в предыдущие десятилетия и возможных последствий для будущего. Основные понятия компьютерной архитектуры были представлены на примере обсуждения регистров и набора инструкций микропроцессора 6502. История компьютерной архитектуры увлекательна, и я рекомендую вам изучить ее более подробно.

В следующей главе мы познакомимся с цифровой логикой, начиная со свойств основных электрических схем и заканчивая проектированием цифровых подсистем, используемых в современных процессорах. Вы узнаете об устройстве логических

вентилей, триггеров и цифровых схем, включая мультиплексоры, регистры сдвига и сумматоры. Эта глава включает введение в языки описания аппаратных средств, которые являются специализированными компьютерными языками, применяемыми при проектировании сложных цифровых устройств, таких как компьютерные процессоры.

Упражнения

1. Используя свой любимый язык программирования, разработайте модель одноразрядного десятичного сумматора, который работает так же, как сумматор аналитической машины Бэббиджа. Сначала запросите у пользователя две цифры в диапазоне 0–9: слагаемое и аккумулятор. Отобразите слагаемое, аккумулятор и признак переноса, который изначально равен 0. Выполните серию циклов следующим образом:

   1) если слагаемое равно 0, выведите на экран значения слагаемого, аккумулятора и признака переноса, после чего завершите программу;

   2) уменьшите слагаемое на 1 и увеличьте значение аккумулятора на 1;

   3) если значение аккумулятора увеличивается с 9 до 0, установите признак переноса;

   4) вернитесь к *шагу 1*.

   Протестируйте свой код на примере следующих операций сложения: 0 + 0, 0 + 1, 1 + 0, 1 + 2, 5 + 5, 9 + 1 и 9 + 9.

2. Создайте для слагаемого, аккумулятора и признаков переноса массивы из 40 десятичных цифр каждый. Запросите у пользователя два целых десятичных числа длиной до 40 цифр каждое. Выполните сложение чисел, цифра за цифрой, используя циклы, описанные в *упражнении 1*, и соберите выходные данные в виде признаков переноса из позиции каждой цифры в массиве признаков переноса. После завершения циклов вставьте переносы и, при необходимости, распространите их по цифрам, чтобы завершить операцию сложения. Выводите результаты на экран после каждого цикла и по завершении. Проведите тест с теми же суммами, что и в *упражнении 1*, а также проверьте суммы 99 + 1, 999999 + 1, 49 + 50 и 50 + 50.

3. Измените программы *упражнений 1* и *2* так, чтобы реализовать вычитание 40-значных десятичных значений. Выполняйте заимствование по мере необходимости. Проверьте программу на примере разностей 0 – 0, 1 – 0, 1000000 – 1 и 0 – 1. Каков результат операции 0 – 1?

4. Язык ассемблера процессора 6502 ссылается на данные в ячейках памяти, используя значение операнда, содержащее адрес (без символа #, который указывает на непосредственное значение). Например, инструкция `LDA $00` загружает в регистр A байт, находящийся в памяти по адресу $00. `STA $01` сохраняет байт из регистра A по адресу $01. Адреса могут иметь любое значение в диапазоне от 0 до $FFFF при условии, что по указанному адресу имеется физическая память и

этот адрес не используется для каких-либо других целей. С помощью предпочтительного эмулятора процессора 6502 напишите код на ассемблере 6502 для сохранения 16-битного значения по адресам $00–$01, сохраните второе значение по адресам $02–$03, затем сложите эти два значения и сохраните результат по адресам $04–$05. Обеспечьте распространение переносов между двумя байтами. Игнорируйте любой перенос из 16-битного результата. Проверьте код на примерах $0000 + $0001, $00FF + $0001 и $1234 + $5678.

5. Напишите на ассемблере 6502 код для вычитания двух 16-разрядных значений способом, аналогичным показанному в *упражнении 4*. Проверьте код на примерах $0001 – $0000, $0001 – $0001, $0100 – $00FF и $0000 – $0001. Каков результат операции $0000 – $0001?
6. Напишите на ассемблере 6502 код для сохранения двух 32-разрядных целых чисел по адресам $00–$03 и $04–$07, а затем сложите их, сохранив результаты по адресам $08–$0B. Используйте циклическую конструкцию, включающую метку и команду ветвления, для перебора байтов двух складываемых значений. Найдите в Интернете подробную информацию об инструкциях уменьшения на 1 и ветвления для процессора 6502, а также об использовании меток в языке ассемблера. Подсказка: в этом приложении хорошо работает реализованный в процессоре 6502 режим адресации с индексированием нулевой страницы.

2

Цифровая логика

Эта глава основывается на вводных темах, представленных в *главе 1*, и дает четкое представление о цифровых структурных элементах, используемых при проектировании современных процессоров и других сложных электронных схем. Мы начнем с обсуждения базовых элементов электрических схем. Далее познакомимся с транзисторами и исследуем их применение в качестве переключающих устройств в простых логических вентилях. Затем мы построим из логических вентилей защелки, триггеры и кольцевые счетчики. После этого путем объединения представленных ранее устройств создадим более сложные компоненты процессора, такие как регистры и сумматоры. Также здесь будет введено понятие последовательностной логики, т. е. логики, построенной на информации о состоянии, которая меняется со временем. Глава заканчивается введением в языки описания аппаратных средств, представляющие предпочтительный метод проектирования сложных цифровых устройств.

В этой главе будут рассмотрены следующие темы:

- электрические схемы;
- транзистор;
- логические вентили;
- защелки;
- триггеры;
- регистры;
- сумматоры;
- синхронизация;
- последовательностная логика;
- языки описания аппаратных средств.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Электрические схемы

Мы начнем эту главу с краткого обзора свойств электрических цепей.

Проводящие материалы, такие как медь, обладают способностью легко создавать электрический ток в присутствии электрического поля. Непроводящие материалы, например стекло, резина и **поливинилхлорид** (ПВХ), препятствуют прохождению электрического тока настолько сильно, что их используют в качестве изоляторов для защиты электрических проводников от коротких замыканий. В металлах электрический ток состоит из движущихся электронов. В конструкции резисторов используются материалы, которые допускают протекание некоторого электрического тока, но при этом предсказуемо ограничивают его допустимую величину.

Связь между силой электрического тока, напряжением и сопротивлением в цепи аналогична связи между скоростью потока, давлением и ограничением потока в гидравлической системе. Рассмотрим кухонный водопроводный кран: давление в трубе, ведущей к крану, заставляет воду течь, когда кран открыт. Если кран приоткрыть совсем чуть-чуть, вода будет течь из него тонкой струйкой. Если кран открыть больше, скорость потока увеличится. Открывание крана эквивалентно уменьшению сопротивления потоку воды, протекающему через кран.

В электрической цепи напряжение соответствует давлению в водопроводе. Электрический ток, сила которого измеряется в **амперах** (часто сокращается до "А"), соответствует скорости потока воды через трубу и кран. Электрическое сопротивление соответствует ограничению потока из-за частично открытого крана.

Величины напряжения, тока и сопротивления связаны формулой $V = IR$, где V — напряжение (в вольтах), I — сила тока (в амперах), R — сопротивление (в омах). Другими словами, напряжение на резистивном элементе цепи равно произведению протекающего через элемент тока на его сопротивление. Это **закон Ома**, названный в честь Георга Ома, впервые опубликовавшего эту формулу в 1827 г.

На рис. 2.1 показано простое схематическое представление этого отношения. Стопка горизонтальных линий слева обозначает источник напряжения, например аккумулятор или блок питания компьютера. Зигзагообразная форма справа представляет собой резистор. Линии, соединяющие компоненты схемы, — это провода, которые считаются идеальными проводниками. Идеальный проводник обеспечивает протекание электрического тока без сопротивления.

Ток, обозначенный буквой I, течет по цепи по часовой стрелке, от положительного полюса батареи, через резистор и обратно к отрицательному полюсу батареи. От-

рицательная сторона батареи определяется в этой схеме как опорная точка с нулевым напряжением.

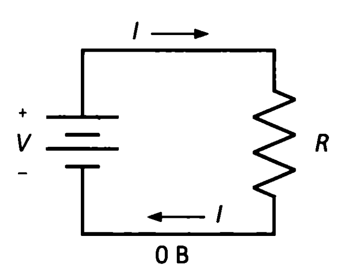


***Рис. 2.1.** Простая резистивная схема*

По аналогии с водопроводной трубой провод под нулевым напряжением представляет собой "бассейн с водой". "Насос" (батарея на схеме) забирает воду из бассейна и выталкивает ее из верхней части символа батареи в трубу под более высоким давлением. Вода течет в виде тока I к крану, обозначенному резистором R справа. После прохождения через кран, ограничивающий поток, вода попадает в бассейн, где ее можно снова набрать насосом.

Если мы предположим, что напряжение батареи (т. е. повышение давления в водяном насосе) постоянно, то любое увеличение сопротивления R уменьшит ток I на обратно пропорциональную величину. Например, удвоение сопротивления снизит ток вдвое. А удвоение напряжения за счет последовательного включения двух батарей, как это делают в фонариках, удвоит ток через резистор.

В следующем разделе мы познакомимся с транзистором, который служит основой для всех современных цифровых электронных устройств.

Транзистор

Транзистор — это полупроводниковое устройство, которое, для целей данного обсуждения, функционирует как цифровой переключатель. **Полупроводник** — это материал, который занимает промежуточное положение по проводимости между хорошими проводниками (такими как медная проволока) и хорошими изоляторами (такими как стекло или пластик). При наличии подходящей схемы проводимость полупроводникового устройства можно изменять с помощью управляющего входа. Используемый таким образом транзистор становится цифровым переключающим элементом.

Операция переключения транзистора электрически эквивалентна переключению между очень высоким и очень низким сопротивлением в зависимости от состояния сигнала на управляющем входе. Важная особенность переключающих транзисторов заключается в том, что для управляющего входа не требуется большой ток.

Это означает, что малый ток на управляющем входе может управлять коммутацией гораздо большего тока, проходящего через транзистор. Ток на выходе одного транзистора может управлять входами многих других транзисторов. Эта особенность очень важна для разработки сложных цифровых схем.

На рис. 2.2 показано схематическое обозначение транзистора типа *n-p-n*. Обозначение *n-p-n* относится к строению взаимосвязанных областей кремниевого кристалла, составляющих транзистор. В область *n* кристалла с помощью процесса, называемого **легированием**, добавляется материал, увеличивающий количество доступных электронов. Область *p* легируется материалом, уменьшающим количество доступных электронов. Транзистор типа *n-p-n* содержит две области *n*, между которыми находится область *p*. Три вывода устройства подсоединены к каждой из этих областей.

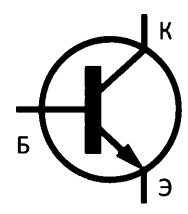


Рис. 2.2. *Схематическое обозначение транзистора типа n-p-n*

Коллектор, обозначенный буквой К на рис. 2.2, подсоединен к одной из *n* областей, а эмиттер Э — к другой области *n*. База Б подсоединена к области *p* между двумя областями *n*. Коллектор "собирает" ток, а эмиттер "испускает" ток в направлении, указанном стрелкой. Вывод базы является управляющим входом. Изменяя напряжение, подаваемое на вывод базы, и, таким образом, изменяя величину тока, протекающего через базу, можно регулировать ток, входящий через коллектор и выходящий через эмиттер.

Логические вентили

На рис. 2.3 представлена принципиальная схема транзисторного вентиля НЕ. Питание схемы обеспечивает источник напряжением 5 В. Входной сигнал может поступать от схемы с кнопкой, которая выдает 5 В, когда кнопка нажата, и 0 В, когда кнопка отпущена. Резистор R_1 ограничивает ток, протекающий от клеммы входа к выводу базы транзистора, когда на входе высокий уровень (около 5 В). В типичной схеме сопротивление резистора R_1 составляет около 1000 Ом. Резистор R_2 может иметь сопротивление 5000 Ом. Он ограничивает ток, протекающий от коллектора к эмиттеру, когда транзистор включен (открыт).

На клемму входа поступает входное напряжение в диапазоне от 0 до 5 В, но поскольку мы говорим о работе цифровой схемы, нас интересуют только сигналы, близкие к 0 В (низкий уровень) или 5 В (высокий уровень). Будем считать, что все уровни напряжения между низким и высоким являются переходными при почти мгновенных переходах между низким и высоким уровнями.

Типичный *n-p-n*-транзистор имеет напряжение переключения около 0,7 В. Когда на клемме входа поддерживается низкое напряжение, например 0,2 В, транзистор закрыт и имеет очень большое сопротивление между коллектором и эмиттером. При этом резистор R_2, подсоединенный к источнику питания 5 В, обеспечивает высокий уровень сигнала на выходе (около 5 В).

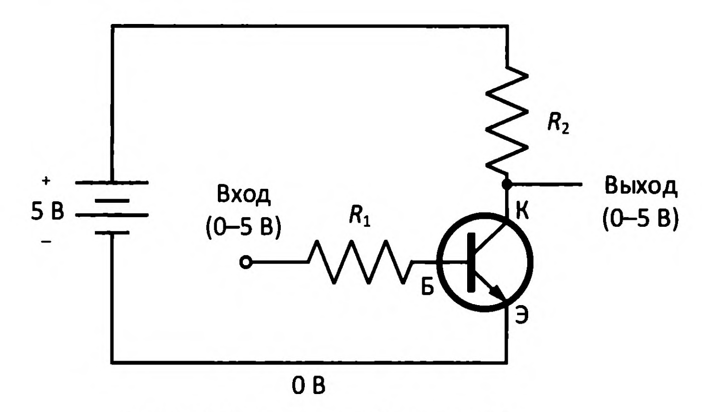


Рис. 2.3. *Транзисторный вентиль НЕ*

Когда напряжение входного сигнала превышает 0,7 В и достигает значения из диапазона от 2 до 5 В, транзистор открывается, и сопротивление между коллектором и эмиттером становится очень маленьким. То есть, по сути, клемма выхода соединяется с проводником 0 В через цепь с сопротивлением, которое намного меньше, чем R_2. В результате на клемме выхода действует низкое напряжение, обычно около 0,2 В.

В общем поведение этой схемы можно выразить так: когда на клемме входа высокий уровень, на клемме выхода наблюдается низкий уровень. Когда на клемме входа низкий уровень, на клемме выхода — высокий уровень. Эта функция описывает вентиль НЕ, в котором выход является инверсией входа. Поведение этого вентиля описано в таблице истинности, представленной в табл. 2.1 (низкому уровню сигнала соответствует двоичное значение 0, а высокому уровню — значение 1).

Таблица 2.1. *Таблица истинности вентиля НЕ*

Выход	Вход
1	0
0	1

Таблица истинности — это табличное представление выходного значения логического выражения как функции всех возможных комбинаций входных данных. Каждый столбец представляет один вход или выход, выходы показаны в правой части таблицы. Каждая строка представляет один набор входных значений вместе с выходным значением выражения с учетом этих входных значений.

Такие схемы, как вентиль НЕ, показанный на рис. 2.3, настолько распространены в цифровой электронике, что им присвоены специальные схематические обозначения для построения диаграмм более высокого уровня, представляющих более сложные логические функции.

Символ вентиля НЕ представляет собой треугольник с маленьким кружком на выходе (рис. 2.4).

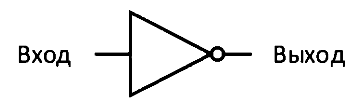


Рис. 2.4. *Схематическое обозначение вентиля НЕ*

Треугольник обозначает усилитель, указывая на то, что это устройство превращает слабый входной сигнал в более сильный выходной сигнал. Кружок обозначает оператор инверсии, который преобразует сигнал в его двоичную противоположность.

Далее мы рассмотрим некоторые более сложные логические операции, которые можно реализовать, опираясь на схему вентиля НЕ. Схема на рис. 2.5 использует два транзистора для выполнения операции И с двумя входами: $Вход_1$ и $Вход_2$. Операция И дает на выходе 1, когда на обоих входах 1, в иных случаях выход равен 0. Резистор R_2 поддерживает низкий сигнал на выходе, если высокие уровни сигналов на входах $Вход_1$ и $Вход_2$ не отпирают оба транзистора.

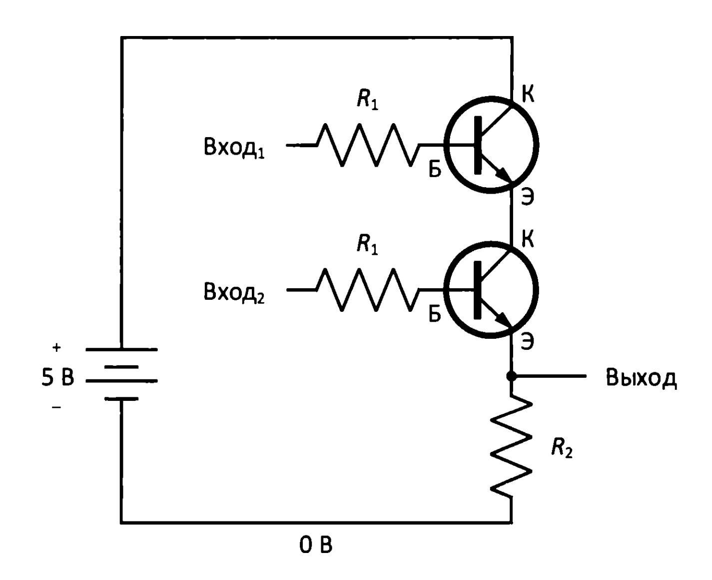


Рис. 2.5. *Транзисторный вентиль И*

В табл. 2.2 представлена таблица истинности вентиля И. Попросту говоря, выходной сигнал является истинным (имеет уровень 1), когда оба входа $Вход_1$ и $Вход_2$ являются истинными, и ложным (0) в прочих случаях.

Таблица 2.2. Таблица истинности вентиля И

Вход$_1$	Вход$_2$	Выход
0	0	0
1	0	0
0	1	0
1	1	1

Вентиль И имеет собственное схематическое обозначение, показанное на рис. 2.6.

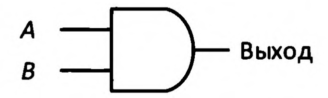


Рис. 2.6. Схематическое обозначение вентиля И

Вентиль ИЛИ выдает на выходе 1, когда любой из входов, *A* или *B*, имеет значение 1, а также когда оба входа имеют значение 1. Таблица истинности вентиля ИЛИ приведена в табл. 2.3.

Таблица 2.3. Таблица истинности вентиля ИЛИ

A	*B*	Выход
0	0	0
1	0	1
0	1	1
1	1	1

Схематическое обозначение вентиля ИЛИ показано на рис. 2.7.

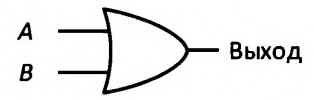


Рис. 2.7. Схематическое обозначение вентиля ИЛИ

Операция исключающего ИЛИ выдает 1, когда только один из входов, *A* и *B*, имеет значение 1. Выход равен 0, когда оба входа равны 0, а также когда оба входа равны 1. Таблица истинности вентиля исключающего ИЛИ приведена в табл. 2.4.

Схематическое обозначение вентиля исключающего ИЛИ показано на рис. 2.8.

Таблица 2.4. *Таблица истинности вентиля исключающего ИЛИ*

A	*B*	Выход
0	0	0
1	0	1
0	1	1
1	1	0

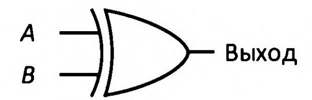


Рис. 2.8. *Схематическое обозначение вентиля исключающего ИЛИ*

Каждый из вентилей — И, ИЛИ и исключающее ИЛИ — можно реализовать с инвертируемым выходом. При этом функция вентиля такая же, как описано выше, за исключением инверсии выхода (в столбце "Выход" табл. 2.2–2.4 ноль заменяется единицей, а единица — нулем). Схематическое обозначение вентилей И, ИЛИ или исключающего ИЛИ с инвертированным выходом имеет небольшой кружок, добавленный на стороне выхода, как и в случае с вентилем НЕ. Названия вентилей с инвертированными выходами И-НЕ, ИЛИ-НЕ и исключающее ИЛИ-НЕ. *НЕ* в каждом из этих названий означает НЕТ. Например, И-НЕ означает инверсию И, что функционально эквивалентно вентилю И, за которым следует вентиль НЕ.

Простые логические вентили можно комбинировать для получения более сложных функций. Мультиплексор — это схема, которая выбирает один из нескольких входов для передачи на свой выход в зависимости от состояния входа селектора. На рис. 2.9 показана схема мультиплексора с двумя входами.

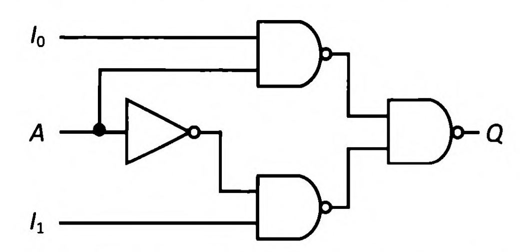


Рис. 2.9. *Схема мультиплексора с двумя входами*

Схема имеет два однобитных входа данных: I_0 и I_1. Когда на входе селектора *A* высокий уровень, на выход *Q* передается значение I_0. Когда на входе *A* низкий уровень, на выход передается значение I_1. Одно из применений мультиплексора в архитектуре процессора — выбор входных данных одного из нескольких источников при загрузке внутреннего регистра.

Таблица истинности мультиплексора с двумя входами приведена в табл. 2.5. В этой таблице значение X означает "безразлично", т. е. состояние этого сигнала не влияет на определение состояния выхода Q.

Таблица 2.5. *Таблица истинности мультиплексора с двумя входами*

A	I_0	I_1	Q
1	0	X	0
1	1	X	1
0	X	0	0
0	X	1	1

Логические вентили, описанные в этом разделе, и схемы, построенные на их основе, представляют **комбинационную логику**, когда значение на выходе в любой момент времени зависит только от текущего состояния входов. На данный момент мы игнорируем задержку распространения и предполагаем, что выход схемы немедленно реагирует на изменения входов. Другими словами, при этих допущениях выход не зависит от предыдущих значений входов. Комбинационные логические схемы не запоминают прежние значения входов или выходов.

В следующем разделе мы рассмотрим некоторые схемы, которые могут сохранять сведения о предыдущих операциях.

Защелки

Комбинационная логика не позволяет напрямую сохранять данные, что необходимо для реализации цифровых функций, таких как регистры процессора. Логические вентили можно использовать для создания элементов хранения данных путем передачи сигнала обратной связи с выхода одного вентиля на вход другого вентиля, предшествующего первому в цепи передачи сигнала.

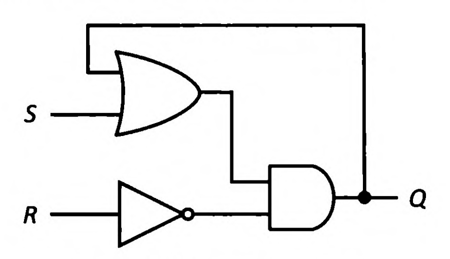


Рис. 2.10. *Схема SR-защелки*

Защелка — это однобитное запоминающее устройство, построенное из логических вентилей. На рис. 2.10 показан простой тип защелки, называемый **защелкой с установкой и сбросом** (set-reset latch) или ***SR*-защелкой**. Особенностью этой схе-

мы, обеспечивающей запоминание состояния, является обратная связь с выхода вентиля И на вход вентиля ИЛИ.

В зависимости от состояния входов S и R эта схема может установить выход Q (перевести его на высокий уровень), сбросить выход Q (перевести его на низкий уровень) или удерживать на выходе Q его последнее значение. В состоянии удержания оба входа S и R имеют низкий уровень, а состояние выхода Q сохраняется. Подача положительного импульса на вход S (переход от низкого уровня к высокому, а затем возврат на низкий уровень) вызывает переход выхода Q на высокий уровень и удержание этого уровня. Подача положительного импульса на вход R вызывает переход выхода Q на низкий уровень и удержание этого уровня. Если на обоих входах, S и R, действует высокий уровень, вход R имеет приоритет над входом S и переводит Q на низкий уровень.

Таблица истинности для RS-защелки приведена в табл. 2.6. $Q_{пред}$ представляет последнее значение выхода Q, выбранное в результате действий входов S и R.

Таблица 2.6. *Таблица истинности RS-защелки*

S	*R*	**Действие**	*Q*
0	0	Удержание	$Q_{пред}$
1	0	Установка	1
X	1	Сброс	0

При работе с этой схемой защелки и вообще с устройствами энергозависимой памяти следует помнить о том, что начальное состояние выхода Q при включении питания является неопределенным. Поведение схемы при запуске и конечное состояние выхода Q зависят от характеристик и синхронизации работы отдельных вентилей по мере их срабатывания. После включения питания и перед началом использования этой схемы по ее назначению необходимо подать импульс на вход S или R, чтобы перевести выход Q в известное состояние.

Управляемая *D*-защелка, где ***D*** означает **данные**, имеет множество применений в цифровых схемах. Термин **"управляемая"** указывает на использование дополнительного входа, который разрешает или запрещает прохождение данных через схему. На рис. 2.11 показана реализация управляемой D-защелки.

Состояние входа D передается на выход Q, когда на входе E (разрешение) высокий уровень. Когда на входе E низкий уровень, выход Q сохраняет свое предыдущее значение независимо от состояния входа D. Выход $\bar{Q}$ всегда выдает значение, обратное выходу Q (горизонтальная черта над $\bar{Q}$ означает НЕ).

Стоит уделить некоторое время прослеживанию логической последовательности этой схемы, чтобы понять ее работу. Левая половина рис. 2.11, состоящая из входа D, вентиля НЕ и двух левых вентилей И-НЕ, представляет собой комбинационную логическую схему, выход которой всегда является прямой функцией входа.

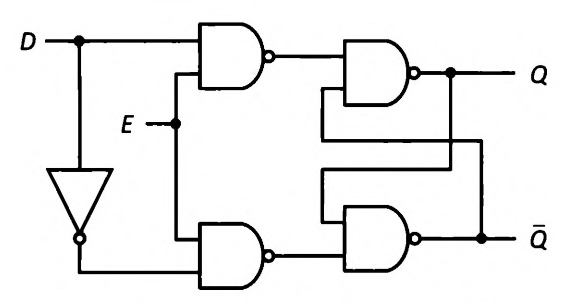


Рис. 2.11. Схема управляемой D-защелки

Таблица 2.7. Таблица истинности управляемой D-защелки

D	E	Q	$\bar{Q}$
0	1	0	1
1	1	1	0
X	0	$Q_{пред}$	$\bar{Q}_{пред}$

Сначала рассмотрим случай, когда на входе E действует низкий уровень. При низком уровне на входе E один из входов каждого из двух левых вентилей И-НЕ имеет низкий уровень, поэтому выход обоих вентилей равен 1 (см. табл. 2.2 и таблицу истинности вентиля И, не забывая, что вентиль И-НЕ эквивалентен вентилю И, за которым следует вентиль НЕ). В этом состоянии значение входа D не имеет значения, и один из выходов, Q или $\bar{Q}$, должен иметь высокое значение, а другой — низкое, из-за перекрестного соединения выходов двух правых вентилей И-НЕ с их входами. Это состояние будет сохраняться до тех пор, пока на входе E действует низкий уровень.

Когда на входе E высокий уровень, то, в зависимости от состояния входа D, на выходе одного из двух левых вентилей И-НЕ будет низкий уровень, а на выходе другого — высокий. Тот из них, на выходе которого действует низкое значение, обеспечит высокий выходной сигнал на подсоединенном к нему правом вентиле И-НЕ. Этот выходной сигнал подается в виде обратной связи на вход другого правого вентиля И-НЕ и при высоком уровне на обоих входах на выходе будет низкий уровень. В результате входной сигнал D передается на выход Q, а его обратное значение появляется на выходе $\bar{Q}$.

Важно понимать, что значения на выходах Q и $\bar{Q}$ не могут быть одновременно высокими или низкими, т. к. такая ситуация представляет собой конфликт между выходами и входами двух правых вентилей И-НЕ. Если одно из этих условий возникнет случайно, например при подаче питания, схема автоматически настроится на стабильную конфигурацию, при которой Q и $\bar{Q}$ будут находиться в противоположных состояниях. Как и в случае с SR-защелкой, результат этой автоматической коррекции непредсказуем, поэтому, прежде чем использовать управляемую D-защелку в каких-либо операциях, необходимо перевести ее в известное исходное состояние.

Инициализация выполняется путем установки высокого уровня на входе *E*, выбора с помощью входа *D* желаемого начального значения на выходе *Q* и затем установки низкого уровня на входе *E*.

Описанная выше управляемая *D*-защелка является **устройством, запускаемым по уровню сигнала**, т. е. значение на выходе *Q* меняется в зависимости от значения на входе *D*, пока на входе *E* поддерживается высокий уровень. В более сложных цифровых схемах важной задачей становится синхронизация нескольких последовательно соединенных элементов схемы, позволяющая избавиться от необходимости тщательно учитывать задержки распространения сигнала между отдельными устройствами. Использование **общего тактового сигнала** в качестве входного сигнала для нескольких элементов обеспечивает такую синхронизацию.

В конфигурации с общим тактовым генератором компоненты обновляют состояния своих выходов в момент прохождения фронта или спада импульса тактового сигнала (т. е. в момент перехода от низкого уровня к высокому или от высокого уровня к низкому), а не непрерывно отслеживают высокий или низкий уровень входного сигнала.

Срабатывание по фронту или спаду импульсов тактового сигнала полезно, поскольку эти события определяют точные моменты, в которые входы устройства должны быть стабильными и достоверными. После прохождения фронта или спада тактового сигнала входы устройства могут произвольно меняться при подготовке к следующему активному фронту или спаду тактового сигнала, не оказывая влияния на уровень сигнала на выходах схемы. Рассмотренная далее схема триггера реагирует на фронты тактовых импульсов, обеспечивая эту желательную характеристику для сложных цифровых схем.

Триггеры

Устройство, которое изменяет состояние выхода только тогда, когда тактовый сигнал совершает заданный переход (от низкого уровня к высокому или от высокого уровня к низкому), называется **устройством, запускаемым по фронту сигнала**. **Триггеры** похожи на защелки, с той лишь разницей, что сигнал на выходе триггера определяется воздействием фронта тактового сигнала, а не непрерывного входного сигнала в течение срока его действия.

***D*-триггер, срабатывающий по фронту**, является популярным компонентом цифровых схем и применяется для решения самых разных задач. В нем обычно используются входные сигналы установки и сброса, которые выполняют те же функции, что и в *SR*-защелке. Этот триггер имеет вход *D*, который функционирует так же, как вход *D* управляемой *D*-защелки. Вместо входа разрешения у *D*-триггера имеется тактовый вход, который запускает передачу сигнала со входа *D* на выход *Q* и, с инверсией, на выход $\bar{Q}$ по фронту тактового сигнала. Триггер не реагирует на изменение состояние входа *D* за пределами очень узкого временно́го окна, окружающего фронт тактового сигнала. Изменения состояния входов *S* и *R* имеют приоритет над любыми изменениями на входе *D* и входе тактового сигнала.

На рис. 2.12 представлено схематическое обозначение *D*-триггера. Вход тактового сигнала обозначен маленьким треугольником в левой части символа.

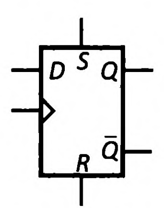


Рис. 2.12. *Схематическое обозначение D-триггера*

Рассмотрим табл. 2.8. Направленные вверх стрелки в столбце *CLK* указывают на фронт тактового сигнала. Значения для выходов Q и $\bar{Q}$ в строках таблицы с направленными вверх стрелками представляют состояние выходов после прохождения фронта тактового сигнала.

Таблица 2.8. *Таблица истинности D-триггера*

S	*R*	*D*	*CLK*	*Q*	$\bar{Q}$
0	0	1	↑	1	0
0	0	0	↑	0	1
0	0	X	Стаб.	$Q_{пред}$	$\bar{Q}_{пред}$
1	0	X	X	1	0
0	1	X	X	0	1

Триггеры можно соединить последовательно, чтобы обеспечить передачу битов данных от одного триггера к другому в последовательных тактовых циклах. Это достигается путем подсоединения выхода *Q* первого триггера ко входу *D* второго триггера с аналогичным продолжением для любого количества ступеней. Эта структура, называемая **регистром сдвига**, имеет множество вариантов применения, включая последовательно-параллельное преобразование и параллельно-последовательное преобразование.

Если выход *Q* в конце регистра сдвига подсоединить к входу *D* на другом конце регистра, получится **кольцевой счетчик**. Кольцевые счетчики используются для таких задач, как построение **конечных автоматов**. Последние реализуют математическую модель, которая всегда находится в одном из ряда четко определенных состояний. Переходы между состояниями происходят, когда значения на входах удовлетворяют требованиям для перехода в другое состояние.

Кольцевой счетчик, представленный на рис. 2.13, имеет четыре позиции. Счетчик инициализируется путем подачи на вход *RST* положительного импульса, т. е. сначала высокого, а затем низкого уровня. Этот импульс устанавливает выход *Q* первого (крайнего левого) триггера в состояние 1, а выходы *Q* остальных триггеров —

в состояние 0. После этого каждый фронт сигнала на входе *CLK* передает 1 бит в следующий по порядку триггер. Четвертый импульс *CLK* передает 1 обратно в крайний левый триггер. В любой момент времени на выходе *Q* одного триггера — 1, а выходы всех остальных триггеров имеют значение 0.

Триггеры — это устройства, запускаемые по фронту импульса, и все они управляются общим тактовым сигналом, что делает эту схему **синхронной**.

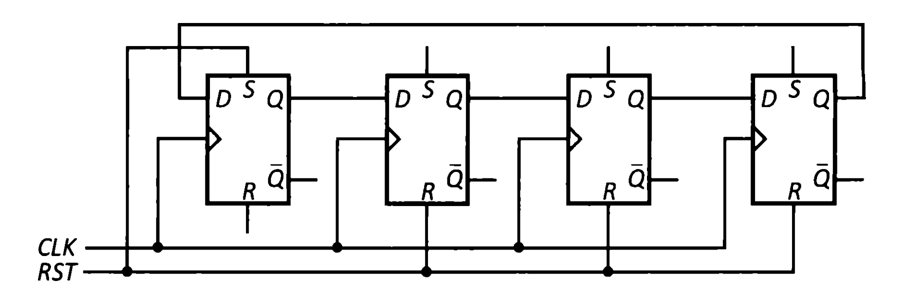


Рис. 2.13. *Схема четырехпозиционного кольцевого счетчика*

Эта схема содержит четыре состояния кольцевого счетчика. Если добавить еще шесть триггеров, то количество состояний достигнет 10. Как мы уже говорили в *главе 1*, в машине ENIAC для сохранения состояния десятичных цифр использовались 10-позиционные кольцевые счетчики на основе электронных ламп. Кольцевой счетчик, хранящий 10 состояний, основанный на схеме, приведенной на рис. 2.13, может выполнять ту же функцию.

В следующем разделе мы создадим из триггеров регистры для хранения данных.

Регистры

Регистры процессора временно хранят значения данных и служат в качестве места хранения входных и выходных данных для операций различных инструкций, включая перемещение данных в память и из памяти, арифметические операции и манипуляции с битами. В большинстве процессоров общего назначения предусмотрены инструкции для сдвига двоичных значений, хранящихся в регистрах, влево или вправо, а также для выполнения операций вращения, при которых биты данных, смещенные с одного конца регистра, вставляются с противоположного конца. Операция вращения аналогична кольцевому счетчику, за исключением того, что биты в цикле могут содержать произвольные значения, в то время как кольцевой счетчик обычно передает один бит, равный единице, через последовательность позиций. Схемы, выполняющие эти функции, строятся из низкоуровневых вентилей и триггеров, представленных ранее в этой главе.

Операции записи и чтения данных регистров внутри процессора обычно выполняются параллельно. Это означает, что все биты записываются или считываются по отдельным сигнальным линиям одновременно под управлением фронта общего тактового сигнала. В примерах, представленных в этом разделе, для простоты ис-

пользуются 4-разрядные регистры, но эти схемы легко расширить до 8, 16, 32 или 64 разрядов.

На рис. 2.14 показан простой 4-разрядный регистр с параллельными вводом и выводом. Это синхронная схема, в которой биты данных, поступающие на входы D_0–D_3, загружаются в триггеры по фронту тактового сигнала *CLK*. Биты данных сразу же появляются на выходах Q_0–Q_3 и сохраняют свое состояние до тех пор, пока новые значения данных не будут загружены по фронту следующего импульса тактового сигнала.

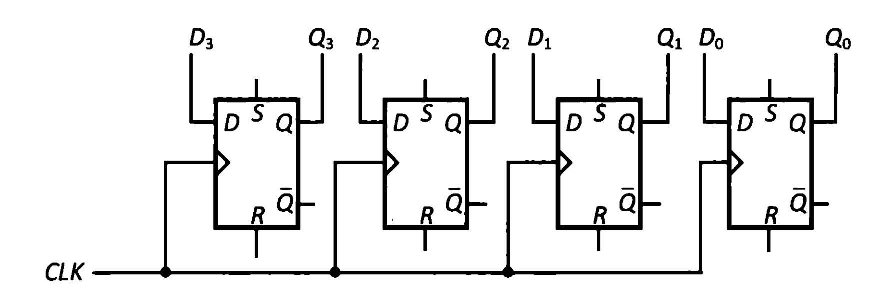


Рис. 2.14. *Схема 4-разрядного регистра*

Для реализации полезных функций, помимо простого хранения данных в регистре, необходима возможность загружать в регистр данные из нескольких источников, выполнять операции с содержимым регистра и записывать получаемые значения в одно из мест назначения, которых может быть несколько.

В процессорах общего назначения данные обычно загружаются в регистр из ячейки памяти, через входной порт или путем переноса из другого регистра. Операции, выполняемые над содержимым регистра, могут включать в себя увеличение на 1, уменьшение на 1, арифметические операции, сдвиг, вращение и битовые манипуляции, такие как операции И, ИЛИ и исключающее ИЛИ. Обратите внимание, что увеличение или уменьшение целого числа на 1 эквивалентно операции сложения или вычитания одного операнда со вторым подразумеваемым операндом, равным 1. После внесения в регистр результата вычисления его содержимое может быть записано в ячейку памяти, в выходной порт или в другой регистр.

На рис. 2.9 показана схема мультиплексора с двумя входами. Эту схему легко расширить, чтобы она поддерживала большее количество входов, любой из которых может быть выбран управляющими сигналами. Однобитовый мультиплексор можно реплицировать для поддержки одновременной работы со всеми битами слова процессора. Такая схема используется для выбора среди множества источников при загрузке данных в регистр. При реализации в процессоре логика, запускаемая опкодами инструкций, устанавливает управляющие входы мультиплексора для маршрутизации данных от выбранного источника в указанный регистр назначения. В *главе 3* использование мультиплексоров будет расширено для маршрутизации данных к регистрам и другим блокам внутри процессора.

В следующем разделе будут представлены схемы для сложения двоичных чисел.

Сумматоры

Процессоры общего назначения обычно поддерживают операцию сложения для выполнения вычислений над значениями данных и отдельно для управления указателем инструкций. После выполнения каждой инструкции указатель инструкций увеличивается на 1 для указания на позицию следующей инструкции.

Если процессор поддерживает инструкции, состоящие из нескольких слов, новое значение указателя инструкций должно равняться своему текущему значению плюс количество слов в только что выполненной инструкции.

Схема простого сумматора складывает два бита данных, учитывает перенос из предыдущего разряда и выдает на выходе однобитную сумму и признак переноса. Эта схема, показанная на рис. 2.15, называется **полным сумматором**, поскольку она учитывает входящий перенос. **Полусумматор** складывает только два бита данных без учета переноса из предыдущего разряда.

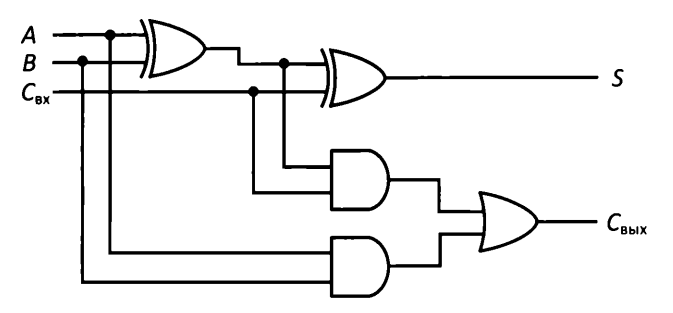


Рис. 2.15. *Схема полного сумматора*

Полный сумматор использует логические вентили для получения результата на выходе следующим образом. Бит суммы S равен 1 только в том случае, если общее количество битов со значением 1 в наборе A, B, $C_{вх}$ является нечетным числом. В ином случае бит S равен 0. Эту логическую операцию выполняют два вентиля типа "исключающее ИЛИ". Выход $C_{вых}$ равен 1, если оба входа, A и B, равны 1, или если только один из входов, A или B, равен 1, и $C_{вх}$ также равен 1. Иначе $C_{вых}$ равен 0.

Для использования в схемах более высокого уровня схему, показанную на рис. 2.15, можно свести к схематическому блоку, который имеет три входа и два выхода. На рис. 2.16 показан 4-разрядный сумматор с четырьмя блоками, представляющими копии схемы полного сумматора с рис. 2.15. Входные данные — это два слова, которые нужно сложить, A_0–A_3 и B_0–B_3, и признак переноса из предыдущего разряда $C_{вх}$. Результатом будут сумма S_0–S_3 и перенос в следующий разряд $C_{вых}$.

Важно отметить, что эта схема является комбинационной. Это означает, что после установки значений на входах выходные значения формируются напрямую. Это также относится и к распространению переноса от бита к биту, независимо от того, на какое количество битов влияют переносы. Поскольку перенос осуществляется по битам, эта конфигурация называется **сумматором со сквозным переносом**.

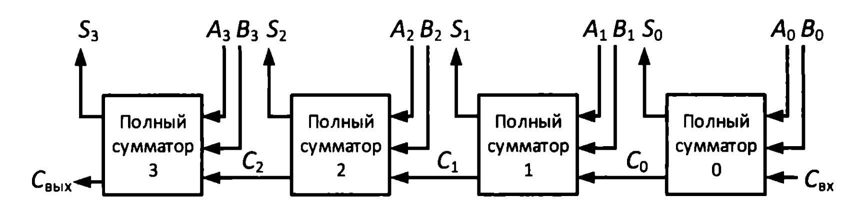


Рис. 2.16. Схема 4-разрядного сумматора

Требуется некоторое время, чтобы переносы распространились по всем позициям битов и на выходах стабилизировались конечные значения.

Поскольку мы сейчас обсуждаем схему, в которой путь сигнала проходит через значительное количество устройств, уместно обсудить факторы, влияющие на время, необходимое для прохождения сигналов через множество компонентов.

Задержка распространения

При изменении входного сигнала логического устройства его выходной сигнал меняется не мгновенно. Между изменением состояния на входе и появлением результата на выходе имеется временна́я задержка. Она называется **задержкой распространения**. Задержка распространения сигнала через схему устанавливает верхний предел тактовой частоты, на которой может работать схема. В микропроцессоре тактовая частота определяет скорость, с которой устройство может выполнять инструкции.

Последовательное соединение нескольких комбинационных схем приводит к общей задержке распространения, равной сумме задержек отдельных устройств. Вентиль может иметь разные значения задержки распространения сигнала для перехода от низкого уровня к высокому и для перехода от высокого уровня к низкому, поэтому при оценке наихудшей задержки в контуре следует использовать большее из этих двух значений.

Как показано на рис. 2.15, самый длинный путь (с точки зрения количества последовательно включенных вентилей) от входа к выходу для полного сумматора — от входов A и B к выходу $C_{вых}$: всего три последовательных вентиля. Если все входные сигналы 4-разрядного сумматора на рис. 2.16 устанавливаются одновременно, задержка в трех вентилях, связанная с входами A и B, будет происходить одновременно во всех четырех сумматорах. Однако выход C_0 полного сумматора 0 будет гарантированно стабильным только после задержки в трех вентилях этого сумматора. После того как C_0 стабилизируется, возникает дополнительная задержка прохождения сигнала через два вентиля полного сумматора 1 (обратите внимание на рис. 2.15, где видно, что сигнал с $C_{вх}$ проходит только через два последовательных уровня вентилей).

Таким образом, общая задержка распространения для схемы на рис. 2.16 составляет три задержки в вентилях на полном сумматоре 0, за которыми следуют по две задержки в вентилях на каждом из остальных трех полных сумматоров — в общей

сложности девять задержек в вентилях. Такая задержка может показаться не слишком значительной, но рассмотрим 32-разрядный сумматор: общая задержка распространения для этого сумматора составляет три задержки в вентилях для полного сумматора 0 плюс две задержки каждого из остальных сумматоров — в общей сложности 65 задержек в вентилях.

Путь с максимальной задержкой распространения сигнала через комбинационную схему называется **критическим путем**. Задержка критического пути устанавливает верхний предел тактовой частоты, которая может использоваться для управления схемой.

Семейство логических вентилей на **усовершенствованных транзисторно-транзисторных логических схемах с барьерами Шотки** (advanced Schottky transistor — transistor logic, (AS) TTL), обозначаемых сокращением **УТТЛШ**, является одним из самых быстрых вентилей в отдельном корпусе, доступных на сегодняшний день.

При типичной нагрузке УТТЛШ-вентиль И-НЕ имеет задержку распространения 2 **наносекунды (нс)**. Для сравнения: за 2 нс свет в вакууме проходит чуть менее 60 см.

В 32-разрядном сумматоре со сквозным переносом 65 задержек распространения сигнала через УТТЛШ-вентили складываются в суммарную задержку между установлением входных сигналов и получением стабильных конечных выходных сигналов, равную 130 нс. Для того чтобы сделать приблизительную оценку, предположим, что это наибольшая возможная задержка распространения сигнала через всю интегральную схему процессора. Мы также проигнорируем любое дополнительное время, требуемое для сохранения стабильности входов до и после активного фронта тактового сигнала. Таким образом, этот сумматор не может выполнять последовательные операции с входными данными чаще, чем раз в 130 нс.

При выполнении 32-разрядного сложения с помощью сумматора со сквозным переносом процессор использует фронт тактового сигнала для переноса содержимого двух регистров (каждый из которых состоит из набора *D*-триггеров) и флага *C* процессора на входы сумматора. Следующий фронт тактового сигнала вызывает загрузку результатов сложения в регистр назначения. Флаг *C* процессора принимает значение $C_{\text{вых}}$ от сумматора.

Тактовый сигнал с периодом 130 нс имеет частоту 1/130 нс, что составляет 7,6 МГц. Эта частота, конечно, не выглядит слишком высокой, особенно если учесть, что сегодня доступно много недорогих процессоров с тактовой частотой более 4 ГГц. Одной из причин этого расхождения является присущее интегральным схемам, содержащим большое количество плотно скомпонованных транзисторов, преимущество в быстродействии, а вторая обусловлена изобретательностью разработчиков, о чем упоминал Гордон Мур (*см. главу 1*). Для эффективного выполнения функций сумматора было разработано множество оптимизированных решений, позволяющих существенно уменьшить наибольшую задержку распространения. В *главе 8* мы обсудим некоторые методы, которые используют архитекторы процессоров, чтобы добиться более высоких скоростей в своих конструкциях.

В дополнение к задержкам вентилей существует также некоторая задержка, возникающая в результате прохождения сигнала по проводам и токоведущим дорожкам интегральной схемы. Скорость прохождения сигнала по проводам или через другой тип проводящего материала меняется в зависимости от свойств материала проводника и изоляционного материала, окружающего проводник. В зависимости от этих и других факторов скорость прохождения сигнала в цифровых схемах обычно составляет 50–90% скорости света в вакууме.

В следующем разделе обсуждается генерирование и использование тактовых сигналов в цифровых схемах.

Синхронизация

Тактовый сигнал задает "сердечный ритм" процессора. Этот сигнал обычно имеет прямоугольную форму и фиксированную частоту. Прямоугольный сигнал — это цифровой сигнал, попеременно принимающий высокое и низкое значения, при этом в каждом цикле пребывание сигнала на высоком и низком уровнях занимает равные промежутки времени. На рис. 2.17 показан пример изменения прямоугольного сигнала во времени.

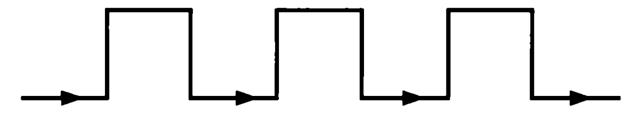


Рис. 2.17. Прямоугольный сигнал

Тактовый сигнал в компьютерной системе обычно генерирует кварцевый генератор с базовой частотой в несколько **мегагерц (МГц)**. 1 МГц — это 1 млн циклов в секунду. Принцип действия кварцевого генератора основан на использовании резонансных колебаний физического кристалла (обычно изготавливаемого из кварца) для формирования циклического электрического сигнала. Механическая вибрация кристалла преобразуется в электрический сигнал с помощью пьезоэлектрического эффекта. **Пьезоэлектрический эффект** — это накопление электрического заряда в определенных кристаллах при механическом воздействии. Кристаллы кварца резонируют на точных частотах, поэтому их используют в качестве элементов синхронизации в компьютерах, наручных часах и других цифровых устройствах.

Кварцевые генераторы являются более точными источниками тактовых сигналов синхронизации, чем альтернативные решения, которые находят применение в недорогих устройствах, однако кристаллам свойственны ошибки в частоте, которые накапливаются в течение нескольких дней и недель, постепенно доводя отклонения от правильного времени до секунд, а затем — до минут. Для того чтобы избежать этой проблемы, большинство компьютеров, подключенных к Интернету, периодически обращаются к серверу времени, чтобы привести свои внутренние часы к текущему времени, публикуемому точными атомными эталонными часами.

Схемы умножителя частоты на основе **контура фазовой автоподстройки частоты** (ФАПЧ) используются для генерирования высокочастотных тактовых сигналов,

необходимых процессорам, работающим на частотах в несколько гигагерц. Умножитель частоты с ФАПЧ генерирует выходной сигнал прямоугольной формы с определенной частотой, которая выражается целым числом, кратным входной частоте, подаваемой на него от кварцевого генератора. Отношение частоты тактового сигнала на выходе умножителя с ФАПЧ к частоте его входного сигнала называется **множителем тактовой частоты**.

Умножитель частоты с ФАПЧ работает путем непрерывной регулировки частоты своего внутреннего генератора для поддержания требуемого множителя тактовой частоты относительно входной частоты ФАПЧ. Современные процессоры обычно имеют вход тактового сигнала кварцевого генератора и содержат несколько умножителей частоты с ФАПЧ, генерирующих разные частоты. Эти выходные частоты ФАПЧ затем управляют основными операциями процессора с максимально возможной скоростью, одновременно взаимодействуя с компонентами, требующими более низких тактовых частот, такими как системная память и периферийные устройства.

Последовательностная логика

Цифровые схемы, генерирующие выходные сигналы на основе комбинации текущих и прежних состояний входов, называются **последовательностными логическими схемами**. Этим она отличается от комбинационной логики, где состояния выходов зависят только от текущего состояния входов.

Если последовательностная логическая схема, состоящая из нескольких компонентов, управляет этими компонентами с помощью общего тактового сигнала, эта схема реализует синхронную логику.

Действия, связанные с выполнением инструкций процессора, представляют собой серию дискретных операций, которые потребляют входные данные в виде опкодов инструкций и значений данных, получаемых из различных источников. Эта деятельность осуществляется при координации на основе главного тактового сигнала. Процессор сохраняет информацию о внутреннем состоянии от одного такта к следующему и от одной инструкции к другой.

Современные сложные цифровые устройства, включая процессоры, почти всегда реализуются как синхронные последовательностные логические устройства. Внутренние компоненты низкого уровня, такие как вентили, мультиплексоры, регистры и сумматоры, рассмотренные ранее, обычно представляют собой комбинационные логические схемы. Эти низкоуровневые компоненты, в свою очередь, получают входные данные под управлением синхронной логики. По прошествии достаточного времени для распространения сигнала по комбинационным компонентам общий тактовый сигнал передает выходные данные этих компонентов другим частям архитектуры под управлением инструкций процессора и логических схем, которые выполняют эти инструкции.

В *главе 3* будут представлены компоненты процессора более высокого уровня, реализующие более сложные функциональные возможности, включая декодирование инструкций, их выполнение и арифметические операции.

В следующем разделе представлена идея проектирования цифрового оборудования с использованием языков, которые очень похожи на традиционные языки программирования компьютеров.

Языки описания аппаратных средств

Простые цифровые схемы легко представить с помощью логических схем, подобных тем, которые были приведены ранее в этой главе. Однако при проектировании более сложных цифровых устройств работа с логическими схемами быстро становится громоздким и неудобным решением. В качестве альтернативы логическим схемам за прошедшие годы было разработано несколько **языков описания аппаратных средств**. Этой разработке способствовал **закон Мура**, который побуждает проектировщиков цифровых систем постоянно находить новые способы наиболее быстрого и эффективного использования постоянно растущего числа транзисторов, доступных в интегральных схемах.

Языки описания аппаратных средств не являются исключительной прерогативой разработчиков цифровых схем в компаниях-производителях полупроводников; даже любители могут приобрести и использовать эти мощные инструменты по доступной цене. Некоторые из них предоставляются бесплатно.

Вентильная матрица — это логическое устройство, содержащее множество логических элементов, таких как вентили И-НЕ и *D*-триггеры, которые можно соединить друг с другом для формирования произвольных схем.

С помощью вентильных матриц, отнесенных к категории **программируемых пользователем вентильных матриц** (ППВМ), конечные пользователи могут реализовывать собственные проекты, используя только компьютер, небольшую макетную плату и соответствующий пакет программного обеспечения.

Разработчик может спроектировать сложную цифровую схему с помощью языка описания аппаратных средств и запрограммировать ее непосредственно в микросхему, в результате чего получится полнофункциональное и высокопроизводительное пользовательское цифровое устройство. Современные недорогие ППВМ содержат достаточное количество вентилей для реализации сложных современных процессорных архитектур. В качестве примера можно привести запрограммированную на ППВМ архитектуру процессора RISC-V (подробно обсуждается в *главе 11*), доступную в виде кода на языке описания аппаратных средств с открытым исходным кодом.

VHDL

VHDL — это один из наиболее распространенных на сегодняшний день языков описания аппаратных средств. Разработка VHDL началась в 1983 г. по заказу Министерства обороны США. Синтаксис и часть семантики VHDL основаны на **языке программирования Ada**. Кстати, язык Ada получил свое название в честь Ады Лавлейс, программиста аналитической машины Чарльза Бэббиджа, обсуждавшейся

в *главе 1*. Другим популярным языком описания аппаратных средств с возможностями, аналогичными VHDL, является Verilog. В этой книге будет использоваться исключительно VHDL, но приведенные здесь примеры также можно легко реализовать на языке **Verilog**.

VHDL — это многоуровневая аббревиатура, в которой "V" расшифровывается как **VHSIC** (very high-speed integrated circuit), что означает **сверхбыстрая интегральная схема** (СБИС), а **VHDL** (VHSIC hardware description language) означает **язык описания аппаратных средств СБИС**. Приведенный ниже код представляет собой реализацию на VHDL схемы полного сумматора, показанной на рис. 2.15.

```
-- Загрузка стандартных библиотек

library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;

-- Определение входов и выходов полного сумматора

entity FULL_ADDER is
  port (
    A     : in   std_logic;
    B     : in   std_logic;
    C_IN  : in   std_logic;
    S     : out  std_logic;
    C_OUT : out  std_logic
  );
end entity FULL_ADDER;

-- Определение поведения полного сумматора

architecture BEHAVIORAL of FULL_ADDER is

begin

  S     <= (A XOR B) XOR C_IN;
  C_OUT <= (A AND B) OR ((A XOR B) AND C_IN);

end architecture BEHAVIORAL;
```

Здесь раздел, начинающийся с `entity FULL_ADDER`, определяет входы и выходы полного сумматора. Раздел `architecture` в конце кода описывает, как работает логика схемы для получения выходных сигналов S и C_OUT на основе входных сигналов

A, *B* и *C_IN*. Термин std_logic указывает на однобитовый двоичный тип данных. Символы <= отражают назначение сигнала, подобное проводному соединению, передающему на выход, указанный в левой части, значение, вычисленное в правой части.

Следующий код ссылается на VHDL-описание полного сумматора FULL_ADDER, как на компонент реализации 4-разрядного сумматора, представленного на рис. 2.16.

```
-- Загрузка стандартных библиотек

library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;

-- Определение входов и выходов 4-разрядного сумматора

entity ADDER4 is
  port (
    A4      : in    std_logic_vector(3 downto 0);
    B4      : in    std_logic_vector(3 downto 0);
    SUM4    : out   std_logic_vector(3 downto 0);
    C_OUT4  : out   std_logic
  );
end entity ADDER4;

-- Определение поведения 4-разрядного сумматора

architecture BEHAVIORAL of ADDER4 is

-- Ссылка на предыдущее определение полного сумматора
  component FULL_ADDER is
    port (
      A     :  in    std_logic;
      B     :  in    std_logic;
      C_IN  :  in    std_logic;
      S     :  out   std_logic;
      C_OUT :  out   std_logic
    );
  end component;

-- Определение внутренних сигналов 4-разрядного сумматора
  signal c0, c1, c2 : std_logic;
```

```
begin

-- Назначение входу переноса первого сумматора значения 0
  FULL_ADDER0 : FULL_ADDER
    port map (
      A       => A4(0),
      B       => B4(0),
      C_IN    => '0',
      S       => SUM4(0),
      C_OUT   => c0
    );

  FULL_ADDER1 : FULL_ADDER
    port map (
      A      => A4(1),
      B      => B4(1),
      C_IN   => c0,
      S      => SUM4(1),
      C_OUT  => c1
    );

  FULL_ADDER2 : FULL_ADDER
    port map (
      A      => A4(2),
      B      => B4(2),
      C_IN   => c1,
      S      => SUM4(2),
      C_OUT  => c2
    );

  FULL_ADDER3 : FULL_ADDER
    port map (
      A      => A4(3),
      B      => B4(3),
      C_IN   => c2,
      S      => SUM4(3),
      C_OUT  => C_OUT4
    );

end architecture BEHAVIORAL;
```

Здесь фрагмент кода, начинающийся с entity ADDER4, определяет входы и выходы 4-разрядного сумматора. Фраза std_logic_vector(3 downto 0) представляет тип данных в виде 4-разрядного вектора с битом № 3 в наиболее значимой позиции и битом № 0 в наименее значимой позиции.

Компонент FULL_ADDER определен в отдельном файле, ссылка на который приведена в начале раздела component FULL_ADDER is. Выражение signal c0, c1, c2 : std_logic; определяет внутренние значения переноса между полными сумматорами. Четыре раздела port map определяют соединения между сигналами 4-разрядного сумматора и входами/выходами каждого из одноразрядных полных сумматоров. Для ссылки на бит в битовом векторе номер бита следует за именем параметра в круглых скобках. Например, *A4(0)* ссылается на наименее значимый из четырех битов в *A4*.

Обратите внимание на использование иерархии в этой структуре. Сначала в отдельном, автономном блоке кода был определен простой компонент — одноразрядный полный сумматор. Затем этот блок был использован для построения более сложной схемы — 4-разрядного сумматора. Такой иерархический подход можно распространить на многие уровни, чтобы определить сложное цифровое устройство, построенное из менее сложных компонентов, каждый из которых, в свою очередь, состоит из еще более простых частей. Этот общий подход обычно используется при разработке современных процессоров, содержащих миллиарды транзисторов, и позволяет контролировать уровень сложности, чтобы архитектура была понятна людям на каждом уровне иерархии.

Код, представленный в этом разделе, содержит все схемные определения, которые требуются программному инструменту логического синтеза для реализации 4-разрядного сумматора в качестве компонента ППВМ. Конечно, необходима дополнительная схема для передачи в схему сумматора осмысленных входных данных, а также для обработки результатов операции сложения с учетом задержки прохождения сигнала.

В этом разделе представлено весьма краткое введение в VHDL. Цель состоит в том, чтобы дать вам понять, что языки описания аппаратных средств, такие как VHDL, являются современным достижением в области проектирования сложных цифровых схем. Кроме того, вам следует знать, что есть некоторые совсем недорогие инструменты и устройства для разработки ППВМ. В упражнениях, приведенных в конце этой главы, вы познакомитесь с некоторыми полезными бесплатными инструментами разработки ППВМ. Рекомендую вам поискать информацию в Интернете, чтобы узнать больше о VHDL и других языках описания аппаратных средств, а также попробовать свои силы в разработке некоторых простых (и не очень простых) схем.

Резюме

Эта глава началась с введения в свойства электрических цепей, которое показало, как компоненты, такие как источники напряжения, резисторы и провода, изображаются на схемах. Был представлен транзистор с акцентом на его использовании в качестве переключающего элемента в цифровых схемах. Из транзисторов и резисторов были составлены схемы вентилей НЕ и И. Были определены логические элементы других типов, и для каждого устройства представлены таблицы истинности. Логические элементы использовались для построения более сложных цифровых схем, включая защелки, триггеры, регистры и сумматоры. Была введена концепция последовательностной логики и обсуждена ее применимость для проектирования процессоров. Наконец, были представлены языки описания аппаратных средств и пример описания 4-разрядного сумматора на языке VHDL.

Теперь вы знакомы с основными концепциями цифровых схем и инструментами проектирования, используемыми при разработке современных процессоров. В следующей главе мы более подробно рассмотрим эти структурные элементы, чтобы изучить функциональные составляющие современных процессоров, а также обсудим, как координируется работа этих компонентов для реализации основного рабочего цикла процессора, состоящего из загрузки, декодирования и выполнения инструкций.

Упражнения

1. Измените схему, представленную на рис. 2.5, чтобы превратить вентиль И в вентиль И-НЕ. Подсказка: добавлять или удалять компоненты не требуется.
2. Создайте реализацию схемы вентиля ИЛИ, изменив схему, представленную на рис. 2.5. По мере необходимости можно добавлять провода, транзисторы и резисторы.
3. Найдите в Интернете бесплатные программные пакеты для разработки на языке VHDL, включающие имитатор. Установите один из этих пакетов, настройте его и попробуйте создать любые простые демонстрационные проекты, включенные в состав пакета, чтобы убедиться, что он работает должным образом.
4. Используя набор инструментов VHDL, создайте 4-разрядный сумматор с помощью листингов, представленных в этой главе.
5. Добавьте в свой 4-разрядный сумматор код тестового драйвера (примеры можно найти в Интернете по фразе *"VHDL testbench"*), затем запустите его с ограниченным набором входных данных и проверьте правильность выходных данных.
6. Раскройте код тестового драйвера и убедитесь, что 4-разрядный сумматор выдает правильные результаты для всех возможных комбинаций входных данных.

3

Элементы процессора

С этой главы начинается развитие нашего всестороннего понимания архитектуры современных процессоров. Опираясь на базовые цифровые схемы, представленные в *главе 2*, мы обсудим функциональные блоки простого процессора для универсального компьютера. В этой главе вводятся понятия, связанные с набором инструкций и набором регистров, за которыми следует обсуждение шагов, связанных с загрузкой, декодированием, выполнением и определением порядка следования инструкций. Режимы адресации и категории инструкций обсуждаются в контексте архитектуры процессора 6502. Мы решили сосредоточиться на этом, заслужившем всеобщее признание процессоре из-за его структурной чистоты и простоты, что позволит нам рассмотреть основные концепции, не отвлекаясь на второстепенные аспекты. Требования к обработке прерываний процессора вводятся на примере обработки прерываний в процессоре 6502. Представлены стандартные подходы, используемые современными процессорами для операций **ввода-вывода** (input/output, I/O), в том числе **прямой доступ к памяти** (direct memory access, DMA).

Прочитав эту главу, вы ознакомитесь с основными компонентами процессора и структурой наборов инструкций. Вы изучите категории инструкций процессора, узнаете, почему необходима обработка прерываний, и получите представление об операциях ввода-вывода.

В этой главе будут рассмотрены следующие темы:

- простой процессор;
- набор инструкций процессора;
- режимы адресации;
- категории инструкций;
- обработка прерываний;
- операции ввода-вывода.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Простой процессор

Архитектура процессора 6502 и несколько его инструкций были представлены в *главе 1*. В данном разделе мы будем опираться на этот фундамент, чтобы представить некоторые функциональные компоненты, широко используемые в процессорных архитектурах, — от самых крошечных встроенных контроллеров до самых мощных серверных процессоров.

Интегральная схема, лежащая в основе компьютерной системы, имеет несколько разных названий: **центральный процессор** (ЦП), микропроцессор или, попросту, процессор. Микропроцессор — это единая интегральная схема, реализующая все функции процессора. В этой книге все категории центральных процессоров и микропроцессоров будут называться процессорами.

Процессор, подобный 6502, содержит три логически различных функциональных блока.

- **Устройство управления** осуществляет общее управление работой процессора. Процесс управления включает в себя извлечение следующей инструкции из памяти, декодирование инструкции для определения операции, которую требуется выполнить, и распределение действий по выполнению инструкции между соответствующими элементами процессора.
- **Арифметико-логическое устройство** (АЛУ) — это комбинационная схема, которая выполняет арифметические операции и операции с битами.
- **Набор регистров** предоставляет исходные и конечные местоположения для хранения входных и выходных данных инструкций. Регистры также используются в качестве мест для временного хранения данных.

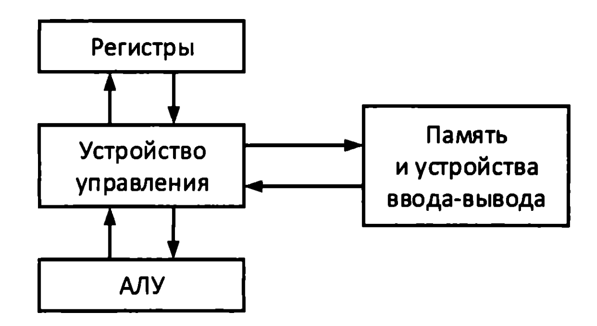


Рис. 3.1. Взаимодействие между функциональными блоками процессора

На рис. 3.1 показаны потоки управления и данных между устройством управления, регистрами, АЛУ, системной памятью и устройствами ввода-вывода.

Устройство управления управляет общими операциями процессора для выполнения каждой инструкции. Регистры, АЛУ, память и устройства ввода-вывода реагируют на команды, инициируемые устройством управления.

Устройство управления

Устройство управления современного процессора представляет собой синхронную последовательностную цифровую схему. Оно интерпретирует инструкции процессора и управляет выполнением этих инструкций, взаимодействуя с другими функциональными блоками внутри процессора и с внешними компонентами, включая память и устройства ввода-вывода. Устройство управления является ключевой частью архитектуры фон Неймана, реализованной в процессоре 6502.

В этой главе термин *"память"* относится к **оперативному запоминающему устройству (ОЗУ, оперативная память)**, являющемуся внешним по отношению к исполнительным блокам процессора. Кеш-память, которая часто располагается на интегральной схеме микропроцессора, будет рассмотрена в следующих главах.

Некоторые примеры устройств ввода-вывода: компьютерная клавиатура, мышь, дисковый накопитель и монитор. Другие распространенные устройства ввода-вывода: сетевые интерфейсы, интерфейсы беспроводной связи Wi-Fi и Bluetooth®, аудиовыход на динамики и микрофонный вход.

После включения компьютерной системы процессор выполняет процедуру сброса для инициализации своих внутренних компонентов. Затем процессор загружает в **программный счетчик** (ПС) адрес ячейки памяти первой инструкции, которая должна быть выполнена. Разработчики программного обеспечения, создающие системные программные компоненты самого низкого уровня, должны настроить свои средства разработки для создания образа кода в памяти, выполнение которого начинается по адресу, требуемому архитектурой процессора.

ПС является центральным компонентом устройства управления. Он всегда содержит адрес инструкции, которая должна быть выполнена следующей. В начале каждого цикла выполнения инструкции устройство управления считывает слово данных из ячейки памяти по адресу, указанному ПС, и помещает его во внутренний регистр для декодирования и выполнения. Первое слово инструкции содержит **код операции** (опкод). На основе битового шаблона опкода устройство управления может считывать дополнительные ячейки памяти, следующие за опкодом, для извлечения данных, требуемых инструкцией, таких как адрес ячейки памяти или операнд данных.

Когда устройство управления начинает выполнять инструкции, оно работает в повторяющемся цикле, показанном на рис. 3.2.

После сброса ПС содержит адрес ячейки памяти с первой инструкцией. Устройство управления извлекает первую инструкцию из памяти и декодирует ее. Во время декодирования устройство управления определяет действия, требуемые инструкцией.

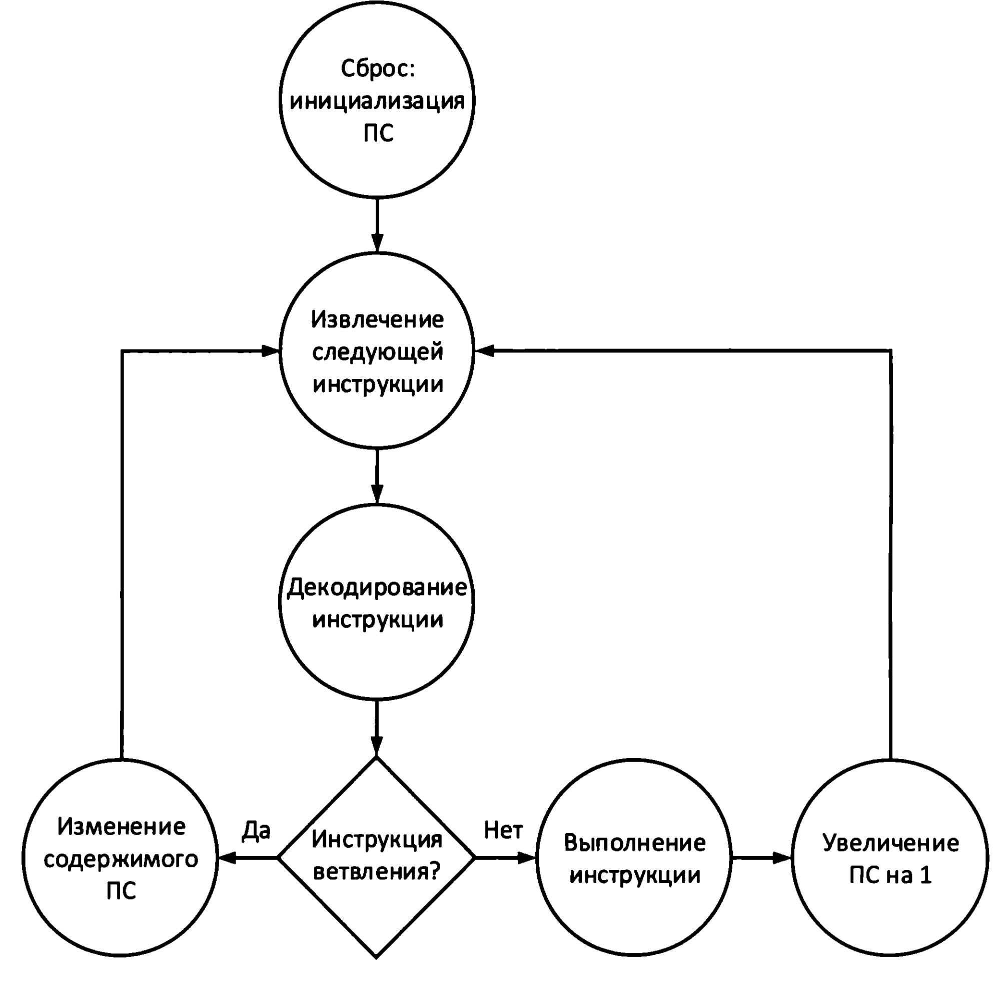


Рис. 3.2. *Цикл выполнения инструкции*

Также в ходе процесса декодирования устройство управления определяет категорию инструкции. На рис. 3.2 представлены две основные категории инструкций — инструкции ветвления и все прочие инструкции. Инструкции ветвления реализуются непосредственно устройством управления. Они вызывают замену содержимого ПС адресом ячейки памяти для выполнения ветви. Примеры инструкций, выполняющих ветвление: инструкции условного ветвления (когда осуществляется выбор ветви), вызовы подпрограмм, возвраты из подпрограмм и инструкции безусловного ветвления (также называемые *переходами*).

Инструкции, не предусматривающие ветвления, выполняются схемой процессора под контролем устройства управления.

В некотором смысле для управления выполнением инструкций без ветвления устройство управления использует способ, аналогичный мельнице аналитической машины (*см. главу 1*), за исключением того, что вместо штифтов на вращающемся барабане для зацепления частей механизма мельницы устройство управления ис-

пользует декодированные биты опкода инструкции для задействования определенных частей цифровой схемы. Выбранные компоненты схемы выполняют задачи, требуемые инструкцией.

Процесс выполнения команды может включать в себя такие действия, как чтение или запись регистра, чтение или запись ячейки памяти, указание АЛУ выполнить математическую операцию и другие действия.

В большинстве процессоров выполнение одной инструкции занимает несколько тактовых циклов. Количество циклов для выполнения инструкции может меняться в широких пределах — от простых инструкций, требующих небольшого количества тактов, до сложных операций, выполнение которых занимает много циклов. Устройство управления координирует всю эту деятельность.

Схемы, управляемые устройством управления, построены из простых логических вентилей, рассмотренных в *главе 2*, и часто включают в себя элементы более высокого уровня, такие как мультиплексоры, защелки и триггеры. Мультиплексоры, в частности, обычно используются логикой устройства управления для выборочной маршрутизации данных в заданное место назначения.

Выполнение инструкции — простой пример

Рассмотрим упрощенный пример выполнения двух инструкций процессора 6502: TXA и TYA. TXA копирует содержимое регистра X в регистр A, а TYA делает то же самое, используя в качестве источника регистр Y. Если мы рассмотрим эти две инструкции отдельно от прочих, то их выполнение может быть реализовано, как показано на рис. 3.3.

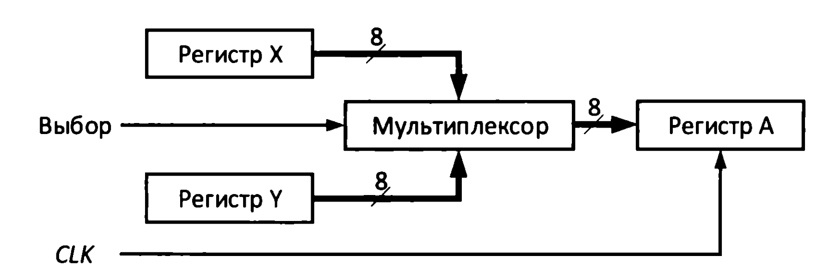


***Рис. 3.3.** Инструкции TXA и TYA процессора 6502*

В схеме на рис. 3.3 предполагается, что регистры X и Y построены на основе *D*-триггеров (как на рис. 2.14), за исключением того, что в процессоре 6502 они представляют собой 8-битные регистры, а не 4-битные. Мультиплексор реализован в виде восьми копий одноразрядного мультиплексора с двумя входами, как на рис. 2.9. Для управления всеми копиями используется один общий вход выбора. На рис. 3.3 более толстые линии представляют 8-разрядные шины данных, а более тонкие линии — отдельные логические сигналы. Короткие штрихи, пересекающие толстые линии с цифрой 8 над ними, указывают количество битов в шине.

Для выполнения инструкции TXA предпринимаются следующие шаги:

1. Сначала устройство управления подает на вход **Выбор** сигнал для передачи на выход мультиплексора битов данных регистра X. Данные из регистра X поступают на вход **регистра A**.
2. После подачи сигнала на вход **Выбор** мультиплексора устройство управления должно выдержать паузу, чтобы биты данных достигли выходов мультиплексора.
3. После стабилизации выходных сигналов мультиплексора устройство управления генерирует сигнал ***CLK***, по фронту которого биты данных регистра X загружаются в регистр A.

Для выполнения инструкции TYA устройство управления выполняет ту же последовательность действий, за исключением того, что сначала оно должно подать на вход **Выбор** сигнал для передачи на выход мультиплексора данных регистра Y.

Это очень простой пример работы устройства управления с инструкциями, но он демонстрирует, что выполнение отдельной инструкции может состоять из нескольких этапов и способно задействовать лишь небольшую часть логических схем, имеющихся в процессоре. Устройство управления должно гарантировать, что компоненты процессора, не используемые для выполнения инструкции, находятся в режиме простоя. Это предотвращает вмешательство указанных компонентов в выполнение инструкции и сводит к минимуму энергопотребление.

Арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ) выполняет арифметические и битовые операции под управлением устройства управления. Для выполнения операции АЛУ требует ввода значений данных, называемых **операндами**, вместе с кодом, указывающим на операцию, которая должна быть выполнена. Выходные данные АЛУ являются результатом операции. Операции АЛУ могут использовать в качестве входных данных один или несколько флагов процессора, таких как флаг переноса, а также устанавливать состояния флагов процессора в качестве выходных данных. В процессоре 6502 операции АЛУ меняют состояния флагов переноса, отрицания, нуля и переполнения.

АЛУ — это комбинационная схема, которая подразумевает, что ее выходы обновляются асинхронно, реагируя на изменения на входах. Кроме того, это устройство не сохраняет данные о предыдущих операциях.

Для того чтобы выполнить инструкцию, затрагивающую АЛУ, устройство управления передает в АЛУ входные данные, выдерживает паузу, чтобы учесть задержку прохождения сигнала по цепям АЛУ, а затем передает выходные данные АЛУ по адресу, указанному инструкцией.

АЛУ содержит схему сумматора для выполнения операций сложения и вычитания. В процессоре с арифметикой в дополнительном коде вычитание можно реализовать путем выполнения отрицания в дополнительном коде для правого операнда и до-

бавления результата к левому операнду. С математической точки зрения, при выполнении вычитания таким образом выражение $A - B$ преобразуется в $A + (-B)$.

Как вы помните из *главы 1*, отрицание в дополнительном коде для числа со знаком выполняется путем инвертирования всех битов в операнде и добавления 1 к результату. С учетом этой операции вычитание, представленное в виде $A + (-B)$, превращается в $A + (\text{NOT}(B) + 1)$.

Рассмотрение вычитания в этой форме должно прояснить использование флага переноса в процессоре 6502 в сочетании с инструкцией SBC. Когда заимствование при вычитании отсутствует, флаг С обеспечивает добавление 1. При наличии заимствования сумма должна быть уменьшена на 1, что достигается путем обнуления флага С.

Подводя итог, можно сказать, что в процессоре 6502 логика вычитания идентична логике сложения с тем единственным отличием, что операнд B в выражении $A - B$ пропускается через набор вентилей НЕ для инвертирования всех восьми битов перед подачей значения НЕ(B) на вход сумматора.

На рис. 3.4 показано функциональное представление операций сложения и вычитания в процессоре 6502.

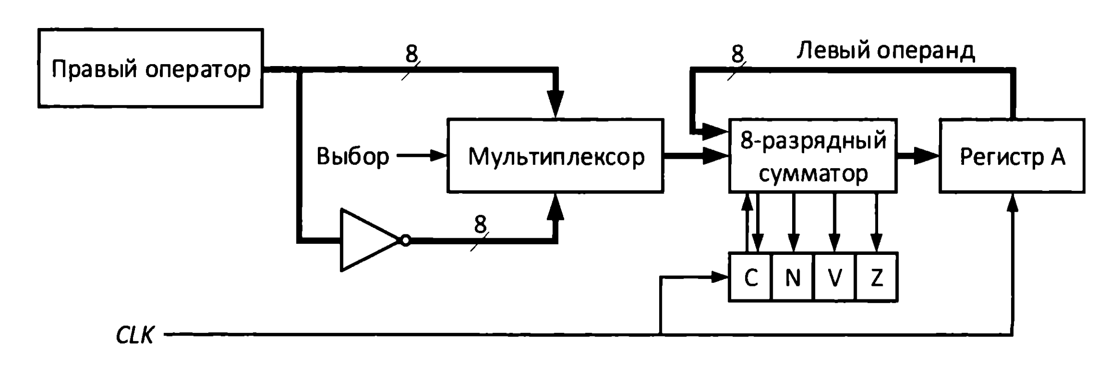


Рис. 3.4. *Операции сложения и вычитания в процессоре 6502*

Как и на рис. 3.3, на рис. 3.4 показано сильно упрощенное представление процессора 6502, где имеются только те компоненты, которые участвуют в выполнении инструкций ADC и SBC. Вход **Выбор** на рис. 3.4 определяет тип операции: сложение или вычитание. Для сложения требуется выбрать верхний вход мультиплексора, а для вычитания — нижний. В архитектуре процессора 6502 регистр A всегда содержит левый операнд для операции вычитания.

Входными данными для сумматора являются левый и правый операнды и флаг С. При выполнении инструкции ADC или SBC устройство управления передает правый операнд на входы данных мультиплексора и устанавливает вход **Выбор** мультиплексора в состояние, соответствующее выполняемой инструкции. После паузы, требуемой для прохождения сигнала через вентиль НЕ, мультиплексор и сумматор, устройство управления генерирует сигнал синхронизации, по фронту которого выходные данные сумматора фиксируются в регистре A и регистре флагов процессора.

Флаги процессора устанавливаются, как показано в следующем описании выполнения инструкции ADC или SBC:

* **C** указывает на наличие переноса при сложении (C = 1) или заимствования при вычитании (C = 0);
* **N** содержит значение бита 7 результата;
* **V** указывает, произошло ли переполнение с учетом знака (V = 1, если произошло переполнение);
* **Z** равен 1, если результат равен нулю; в противном случае Z равен 0.

Помимо сложения и вычитания двух чисел АЛУ поддерживает множество других операций. В процессоре 6502 для большинства операций с двумя операндами левый операнд помещается в регистр A. Правый операнд либо считывается из ячейки памяти, либо предоставляется в виде непосредственного значения в ячейке памяти, следующей после кода операции. В процессоре 6502 все операнды и результаты АЛУ являются 8-битными значениями. Операции АЛУ процессора 6502:

* ADC, SBC — сложение или вычитание двух операндов со входом переноса;
* DEC, DEX, DEY — уменьшение значения в ячейке памяти или регистре на единицу;
* INC, INX, INY — увеличение значения в ячейке памяти или регистре на единицу;
* AND — побитовая логическая операция И над двумя операндами;
* ORA — побитовая логическая операция ИЛИ над двумя операндами;
* EOR — побитовая логическая операция "исключающее ИЛИ" над двумя операндами;
* ASL, LSR — сдвиг значения в регистре A или ячейке памяти влево или вправо на один разряд и вставка 0 в освободившуюся позицию;
* ROL, ROR — вращение значения в регистре A или ячейке памяти влево или вправо на один разряд и вставка значения флага C в освободившуюся позицию;
* CMP, CPX, CPY — вычитание одного операнда из другого и отбрасывание результата с установкой флагов N, Z и C в зависимости от результата вычитания;
* BIT — побитовая логическая операция И над двумя операндами и установка флага Z, чтобы указать, равен ли результат 0; кроме того, копирование битов 7 и 6 левого операнда во флаги N и V.

Возможности АЛУ процессора 6502 ограничены по сравнению с более сложными современными процессорами, такими как семейство *x*86.

Например, в 6502 операции умножения и деления программист должен реализовать в коде в виде повторяющихся операций сложения и вычитания. Кроме того, процессор 6502 может выполнять сдвиг или вращение значения в рамках одной инструкции всего на один разряд.

В процессорах семейства *x*86, с другой стороны, инструкции умножения и деления реализованы непосредственно, а инструкции сдвига и вращения включают параметр, указывающий количество разрядов для сдвига в рамах одной инструкции.

АЛУ — это неизбежно сложное логическое устройство, что делает его идеальным кандидатом для проектирования с помощью языка описания аппаратных средств. Следующий листинг — это реализация на языке VHDL части АЛУ, подобного тому, что применяется в процессоре 6502:

```
-- Загрузка стандартных библиотек
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- Определение входов и выходов 8-разрядного АЛУ
entity ALU is
  port (
    -- Левый операнд
    LEFT        : in     std_logic_vector(7 downto 0);
    -- Правый операнд
    RIGHT       : in     std_logic_vector(7 downto 0);
    -- Операция АЛУ
    OPCODE      : in     std_logic_vector(3 downto 0);
    -- Перенос из предыдущего разряда
    C_IN        : in     std_logic;
    -- Результат АЛУ
    RESULT      : out    std_logic_vector(7 downto 0);
    -- Перенос в следующий разряд
    C_OUT       : out    std_logic;
    -- Вывод флага знака
    N_OUT       : out    std_logic;
    -- Вывод флага переполнения
    V_OUT       : out    std_logic;
    -- Вывод флага нуля
    Z_ OUT      : out    std_logic
  );
end entity ALU;

-- Определение поведения 8-разрядного АЛУ

architecture BEHAVIORAL of ALU is

begin

  P_ALU : process (LEFT, RIGHT, OPCODE, C_IN) is
```

```
    variable result8  : unsigned(7 downto 0);
    variable result9  : unsigned(8 downto 0);
    variable right_op : unsigned(7 downto 0);

  begin

    case OPCODE is

      when "0000" | "0001" => -- Сложение или вычитание

        if (OPCODE = "0000") then
          right_op := unsigned(RIGHT);     -- Сложение
        else
          right_op := unsigned(not RIGHT); -- Вычитание
        end if;

        result9 := ('0' & unsigned(LEFT)) +
                   unsigned(right_op) +
                   unsigned(std_logic_vector'(""& C_IN));
        result8 := result9(7 downto 0);

        C_OUT <= result9(8);               -- Флаг C

        -- Флаг V
        if (((LEFT(7) XOR result8(7)) = '1') AND

            ((right_op(7) XOR result8(7)) = '1')) then
         V_OUT <= '1';
        else
          V_OUT <= '0';
        end if;

      when "0010" =>                      -- Увеличение на 1
        result8 := unsigned(LEFT) + 1;
      when "0011" =>                      -- Уменьшение на 1
        result8 := unsigned(LEFT) - 1;
      when "0101" =>                      -- Побитовая операция И
        result8 := unsigned(LEFT and RIGHT);
      when "0110" =>                      -- Побитовая операция ИЛИ
```

```
        result8 := unsigned(LEFT or RIGHT);
      when "0111" =>                    -- Побитовая операция исключающее ИЛИ
        result8 := unsigned(LEFT xor RIGHT);
      when others =>
        result8 := (others => 'X');
    end case;

    RESULT <= std_logic_vector(result8);

    N_OUT <= result8(7); =>             -- Флаг N

    if (result8 = 0) then               -- Флаг Z
      Z_OUT <= '1';
    else
      Z_OUT <= '0';
    end if;

  end process P_ALU;

end architecture BEHAVIORAL;
```

Этот код определяет простое АЛУ как комбинационную схему с левым операндом, правым операндом, кодом операции и флагом С в качестве входных данных. Выходными данными являются результат операции и флаги C, N, V и Z.

Далее мы рассмотрим назначение и функции регистров процессора.

Регистры

Регистры процессора — это внутренние области хранения данных, которые выполняют функции источников и мест назначения для операций, используемых для выполнения инструкций. Регистры обеспечивают наиболее быстрый доступ к данным в процессоре, но ограничены очень небольшим количеством ячеек памяти из-за их высокой стоимости с точки зрения площади матрицы. Разрядность регистра обычно совпадает с размером слова процессора.

Процессор 6502, как мы уже видели, имеет только три 8-разрядных регистра: A, X, и Y. Процессоры семейства *x*86 имеют шесть 32-разрядных регистров для временного хранения данных: EAX, EBX, ECX, EDX, ESI и EDI. В архитектурах многих процессоров определенные регистры выделяются для передачи входных данных, требуемых определенными инструкциями. Например, в архитектуре *x*86 одна инструкция REP MOVSD перемещает блок данных, длина которого (в словах) хранится в регистре ECX,

начинающийся с адреса источника из регистра ESI, по адресу назначения из регистра EDI.

При разработке новой архитектуры процессора крайне важно найти золотую середину между количеством регистров и количеством и сложностью инструкций, доступных процессору. Если взять кристалл интегральной схемы определенного размера и определенной технологии изготовления (оба этих фактора ограничивают количество транзисторов, доступных для процессора), то добавление большего числа регистров в архитектуру уменьшает количество транзисторов, доступных для выполнения инструкций и других функций. С другой стороны, добавление инструкций со сложными возможностями способно ограничить пространство кристалла, доступное для регистров. Это противоречие между сложностью набора инструкций и количеством регистров выражается в отнесении архитектуры к категории CISC или RISC.

- Процессоры категории **CISC (complex instruction set computer — компьютер с полным набором инструкций)** характеризуются наличием обширного набора инструкций, поддерживающего множество функций, таких как возможность загружать операнды из памяти, выполнять операцию и сохранять результат в памяти, и все это с помощью одной инструкции. В процессоре CISC выполнение инструкции может занять много тактов, пока процессор выполняет все необходимые подзадачи. Упомянутая выше инструкция REPMOVSD является примером инструкции с потенциально длительным временем выполнения. Процессоры CISC, как правило, имеют меньшее количество регистров, отчасти из-за того, что значительную площадь кристалла занимают схемные элементы, реализующие логику набора инструкций. Семейство процессоров *x*86 является классическим примером архитектуры CISC.
- Процессоры категории **RISC (reduced instruction set computer — компьютер с сокращенным набором инструкций)**, напротив, имеют меньшее количество инструкций, каждая из которых выполняет более простые задачи по сравнению с инструкциями процессоров CISC. Выполнение операции со значениями данных, хранящимися в памяти, может потребовать пары инструкций для загрузки двух операндов из памяти в регистры, еще одной инструкции для выполнения операции и заключительной инструкции для записи результата операции в память.

Ключевое различие между CISC и RISC заключается в том, что архитектуры RISC оптимизированы для выполнения отдельных инструкций с очень высокой скоростью. Несмотря на то что в RISC-процессоре для чтения данных из памяти, выполнение операции и записи результата в память требуется на несколько инструкций больше, чем в CISC-процессоре, общее время от начала до конца выполнения тех же операций может быть сопоставимым или даже меньшим в случае RISC-процессора. Примерами категории RISC могут служить архитектуры ARM, которые обсуждаются в *главе 10*, или архитектура RISC-V, рассматриваемая в *главе 11*.

Упрощение набора инструкций в RISC-процессорах оставляет больше места на кристалле для регистров, это означает, что в RISC-процессорах обычно использует-

ся большее количество регистров по сравнению с CISC-процессорами. Архитектура ARM, например, имеет 13 регистров общего назначения, в то время как базовая 32-разрядная архитектура RISC-V — 31 регистр общего назначения.

Большее количество регистров в архитектурах RISC уменьшает потребность в доступе к системной памяти, поскольку для хранения промежуточных результатов доступно больше регистров. Это повышает производительность системы, т. к. доступ к системной памяти занимает значительно больше времени, чем доступ к данным, расположенным в регистрах процессора.

Представьте себе регистр процессора как набор *D*-триггеров, в которых каждый триггер содержит один бит данных регистра. Данные загружаются в каждый из триггеров регистра по общему тактовому сигналу. Данные для записи в регистр могут поступать в триггеры после прохождения через мультиплексор, который выбирает один из потенциально многих источников данных следуя указаниям выполняемой инструкции.

В качестве альтернативы использованию мультиплексора для этой цели инструкция может загружать данные в регистр из шины передачи данных, внутренней по отношению к процессору. В этой конфигурации устройство управления организует работу внутренней шины таким образом, чтобы только нужный источник данных управлял линиями шины в момент прохождения фронта тактового сигнала, который загружает данные в регистр, в то время как всем другим источникам данных, подключенным к шине, запрещено размещать на ней данные.

В следующих разделах будет представлен полный состав набора инструкций процессора и используемые инструкциями режимы адресации.

Набор инструкций процессора

Инструкции, аналогичные тем, что обсуждались выше, реализованы в большинстве архитектур процессоров общего назначения, однако возможности более сложных процессоров расширяют за счет дополнительных категорий инструкций. Более совершенные инструкции, доступные в архитектурах современных процессоров, будут представлены в следующих главах.

Процессоры CISC обычно поддерживают несколько режимов адресации. Они, в свою очередь, предназначены для эффективного доступа к последовательным ячейкам памяти, используемым программными алгоритмами, работающими на процессоре. В следующем разделе описываются режимы адресации инструкций, реализованные в процессоре 6502. Далее будут представлены категории инструкций процессора 6502, большинство из которых применяются в архитектурах современных процессоров. Затем мы рассмотрим специализированные инструкции для обработки прерываний и операций ввода-вывода, включая объяснение функций процессора, которые обеспечивают высокопроизводительные операции ввода и вывода с блоками данных переменного размера.

Режимы адресации

CISC-процессоры поддерживают несколько режимов адресации для инструкций, требующих передачи данных между памятью и регистрами. RISC-процессоры имеют ограниченное число режимов адресации. Каждая архитектура определяет свой набор режимов адресации на основе анализа предполагаемых моделей доступа к памяти, которые программное обеспечение будет использовать в этой архитектуре.

Для того чтобы представить режимы адресации процессора 6502, в этом разделе используется простой пример кода 6502, складывающий четыре байта данных. Для того чтобы избежать излишних деталей, в примере игнорируется любой перенос из 8-битной суммы.

Режим непосредственной адресации

При непосредственной адресации значение операнда следует в памяти сразу за кодом операции. Для первого примера предположим, что нам даны значения четырех байтов для сложения и предложено написать для процессора 6502 программу, решающую эту задачу. Такая постановка задачи позволяет нам вводить байтовые значения непосредственно в наш код. Байты в этом примере — от `$01` до `$04`. Мы будем складывать байты в обратном порядке (от `$04` до `$01`), предполагая дальнейший переход к использованию циклической конструкции, которая будет представлена в данном разделе позже. В этом коде для сложения четырех байтов используется **режим непосредственной адресации**:

```
; Сложение четырех байтов с использованием режима непосредственной адресации
LDA #$04
CLC
ADC #$03
ADC #$02
ADC #$01
```

Обратите внимание, что комментарии на языке ассемблера начинаются с точки с запятой. После выполнения этих инструкций регистр A будет содержать значение `$0A` — сумму четырех байтов, указанных в качестве операндов.

Вспомните *главу 1* — там было сказано, что в языке ассемблера 6502 непосредственному значению предшествует символ #, а символ $ указывает, что значение является шестнадцатеричным. Непосредственно адресованный операнд считывается из ячейки памяти по адресу, следующему за кодом операции инструкции. Непосредственная адресация удобна тем, что при ее использовании нет необходимости резервировать ячейку памяти, из которой следует считать операнд. Однако режим непосредственной адресации полезен только в том случае, если значение данных известно во время написания программы.

Режим абсолютной адресации

В **режиме абсолютной адресации**, иногда называемом **режимом прямой адресации**, указывается ячейка памяти, содержащая значение, которое будет считано или записано инструкцией. В процессоре 6502 предусмотрены 16 адресных битов, поэтому адресное поле, поддерживающее доступ ко всей доступной памяти, имеет длину 2 байта. Полная инструкция для доступа к данным в произвольной ячейке памяти процессора 6502 состоит из трех байтов: первый байт — это код операции, за ним следуют два байта с адресом ячейки памяти для считывания или записи. При сохранении двух байтов адреса первым располагается младший байт, а за ним следует старший байт. Старший байт 16-разрядного значения содержит 8 старших значащих битов, а младший байт — 8 младших значащих битов.

Благодаря принятому соглашению о хранении младшего байта двухбайтового адреса по меньшему адресу памяти 6502 является процессором **с прямым порядком байтов** (little-endian). В процессорах семейства *x*86 также принят прямой порядок байтов. Архитектуры ARM и RISC-V позволяют программным образом выбирать режим прямого или обратного порядка следования байтов (**переключаемый порядок байтов** (bi-endianness)), но большинство операционных систем, работающих на этих архитектурах, выбирают режим с прямым порядком байтов.

Пример режима абсолютной адресации мы начнем с некоторого подготовительного кода для сохранения четырех байтов, которые будут складываться, по адресам от `$200` до `$203`. За подготовительным кодом следуют инструкции для сложения этих четырех байтов. В этом примере для сложения четырех байтов применяется режим абсолютной адресации:

```
; Инициализация данных в памяти
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
; Сложение четырех байтов с использованием режима абсолютной адресации
LDA $0203
CLC
ADC $0202
ADC $0201
ADC $0200
```

В отличие от режима непосредственной адресации, абсолютная адресация позволяет суммировать четыре значения, которые не известны до выполнения программы:

инструкции ADC складывают любые значения, сохраненные в ячейках $200–$203. Ограничение этого режима адресации заключается в том, что при написании программы необходимо указать адреса складываемых байтов. Этот код не может складывать байты, расположенные в произвольном месте в памяти.

Недостатком нашего простого примера является использование избыточной последовательности почти идентичных инструкций. Для того чтобы этого избежать, обычно желательно поместить повторяющуюся часть кода в циклическую конструкцию. В следующих двух примерах используется режим адресации 6502, который облегчает выполнение циклических операций, хотя мы не будем добавлять в код цикл до второго примера.

Режим абсолютной индексной адресации

В **режиме абсолютной индексной адресации** адрес ячейки памяти вычисляется путем добавления к базовому адресу, указанному в инструкции, значения, содержащегося в регистре X или Y. В следующем примере с помощью абсолютной индексной адресации складываются байты по адресам от $0200 до $0203. Регистр X предоставляет смещение от начала массива байтов по адресу $0200:

```
; Инициализация данных в памяти
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
; Сложение четырех байтов с использованием режима абсолютной индексной адресации
LDX #$03
CLC
LDA $0200, X
DEX
ADC $0200, X
DEX
ADC $0200, X
DEX
ADC $0200, X
```

Инструкция DEX уменьшает значение в регистре X на единицу (вычитает из него 1). Этот код ухудшает решение с точки зрения увеличения количества инструкций,

необходимых для сложения четырех байтов, однако мы видим, что последовательность из инструкции DEX и следующей за ней инструкции ADC $0200, X теперь повторяется три раза.

Мы можем использовать условное ветвление для выполнения этой последовательности сложения в цикле:

```
; Инициализация данных в памяти
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
; Сложение четырех байтов с использованием режима абсолютной индексной адресации
LDX #$03
LDA $0200, X
DEX
CLC
ADD_LOOP:
ADC $0200, X
DEX
BPL ADD_LOOP
```

Инструкция BPL (ветвление при плюсе) осуществляет условную передачу управления инструкции, перед которой стоит метка ADD_LOOP. BPL выполняет это ветвление только тогда, когда флаг N процессора равен нулю. Если флаг N установлен, инструкция BPL передает управление следующей инструкции в памяти.

Может показаться, что код в этом примере не стоил усилий, затраченных на создание цикла лишь для того, чтобы сложить четыре байта. Но обратите внимание, что эту версию кода можно изменить для сложения 100 последовательных байтов простым изменением операнда инструкции LDX. Изменение предыдущего примера для сложения 100 байтов таким же образом потребовало бы намного больше работы, и инструкции потребляли бы гораздо больше памяти.

Этот пример имеет те же ограничения, что и пример для режима абсолютной адресации — в обоих случаях задается начало массива байтов в ячейке памяти, определенной во время написания программы. Следующий режим адресации устраняет это ограничение и складывает массив байтов, начиная с любого адреса в памяти.

Режим косвенной индексной адресации

Режим косвенной индексной адресации использует двухбайтовый адрес, хранящийся в диапазоне адресов памяти $00–$FF, в качестве базового адреса и добавляет содержимое регистра Y к этой базе для получения адреса памяти, используемого инструкцией. В следующем примере базовый адрес массива байтов ($0200) предварительно сохраняется по адресам $0010 и $0011 с соблюдением прямого порядка байтов. Следующий код использует для сложения байтов косвенную индексную адресацию в цикле:

```
; Инициализация данных в памяти
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
; Инициализация указателя на массив байтов
LDA #$00
STA $10
LDA #$02
STA $11
; Сложение четырех байтов с использованием режима косвенной индексной адресации
LDY #$03
LDA ($10), Y
DEY
CLC
ADD_LOOP:
ADC ($10), Y
DEY
BPL ADD_LOOP
```

При косвенной индексной адресации перед выполнением кода сложения можно сохранить любой адрес в памяти по адресам $10–$11. Обратите внимание, что при косвенной индексной адресации для хранения смещения адреса необходимо использовать регистр Y. Регистр X недоступен для применения в этом режиме адресации.

В процессоре 6502 предусмотрены и некоторые иные режимы адресации: **режим адресации по нулевой странице** (абсолютной и абсолютной с индексированием)

позволяет использовать инструкции, которые имеют меньшую длину (на один байт) и выполняются быстрее за счет работы только с адресами памяти в диапазоне $00–$FF. Термин *"нулевая страница"* связан с тем, что для адресов в этом диапазоне старший байт 16-разрядного адреса равен нулю. Помимо улучшения производительности за счет более высокой скорости выполнения и уменьшения занимаемой кодом памяти, режимы адресации по нулевой странице действуют так же, как и соответствующие режимы адресации, описанные ранее.

Другой режим называется **режимом индексной косвенной адресации**[1] — он похож на режим косвенной индексной адресации, за исключением того, что вместо регистра Y используется регистр X, а смещение, содержащееся в регистре X, добавляется к адресу, указанному в инструкции, для определения адреса указателя на данные. Например, предположим, что регистр X содержит значение 8. Инструкция `LDA ($10, X)` добавляет `$10` к содержимому регистра X, что дает в результате `$18`. Затем эта инструкция использует 16-разрядный адрес памяти, считанный из памяти по адресам `$18–$19`, в качестве адреса целевой ячейки памяти для загрузки значения в регистр А.

Индексная косвенная адресация не имеет отношения к нашему примеру суммирования последовательности байтов. Одним из примеров применения этого режима является выбор значения из последовательного списка указателей, где каждый указатель содержит адрес строки символов. Регистр X может ссылаться на одну из этих строк как на смещение от начала списка указателей. Такая инструкция, как `LDA ($10, X)`, загружает в регистр А адрес выбранной строки.

Режимы адресации, доступные в архитектурах процессоров CISC и, в меньшей степени, в архитектурах RISC, обеспечивают поддержку эффективных методов доступа к различным типам структур данных в системной памяти.

В следующем разделе обсуждаются категории инструкций, реализованные в архитектуре процессора 6502, а также использование каждой инструкцией доступных режимов адресации.

Категории инструкций

В этом разделе представлены категории инструкций, реализованные в процессоре 6502. Цель обсуждения процессора 6502 здесь состоит в том, чтобы представить концепции, связанные с набором инструкций более простой процессорной архитектуры, чем у современных 32- и 64-разрядных процессоров, которые мы рассмотрим в следующих главах. К тому времени, когда мы доберемся до этих процессоров, базовые концепции набора инструкций должны быть вам хорошо знакомы.

---

[1] Режимы индексной косвенной адресации и косвенной индексной адресации названы по аналогии с английскими названиями, выбранными автором ("indexed indirect addressing" и "indirect indexed addressing"), их названия не отражают отличий между ними. — *Пер.*

Инструкции загрузки и сохранения

Процессор 6502 использует инструкции загрузки из памяти и сохранения в память для считывания значений данных из системной памяти в регистры процессора и для записи содержимого регистров в системную память. В архитектуре 6502 инструкции LDA, LDX и LDY загружают в регистр, указанный в их названии, 8-разрядное слово из системной памяти. Инструкция LDA поддерживает все режимы адресации, доступные в 6502, а LDX и LDY — лишь часть режимов адресации: непосредственную, абсолютную и абсолютную индексную.

После завершения выполнения каждой из этих инструкций флаги N и Z указывают, является ли загруженное значение отрицательным (по установленному биту 7) и равно ли это значение нулю.

Инструкции STA, STX и STY записывают в память содержимое регистра, указанного в их названиях. Каждая инструкция сохранения поддерживает те же режимы адресации, что и инструкция загрузки для этого регистра, за исключением режима непосредственной адресации. Эти инструкции обновляют состояние флагов N и Z, отражая свойства сохраненного значения.

Инструкции передачи данных из регистра в регистр

Эти инструкции копируют 8-битное слово одного регистра (A, X или Y) в другой регистр. Данные инструкции используют **режим неявной адресации**, т. е. источник и адресат каждой инструкции указываются непосредственно кодом операции инструкции.

Инструкция TAX копирует содержимое регистра A в регистр X. Инструкции TAY, TXA и TYA выполняют аналогичные операции между парами регистров, указанными в их названиях. Эти инструкции обновляют состояние флагов N и Z.

Инструкции стека

Инструкция TXS копирует содержимое регистра X в регистр указателя стека (S). Она должна использоваться для инициализации регистра S во время запуска системы. Инструкция TSX копирует содержимое регистра S в регистр X и обновляет состояние флагов N и Z. Выполнение инструкции TXS не оказывает влияния на какие-либо иные флаги.

Инструкция PHA помещает в стек содержимое регистра A. Инструкция PHP помещает в стек флаги процессора в виде 8-битного слова. Эти инструкции не влияют на флаги процессора. Помещение значения в стек состоит из записи содержимого регистра в ячейку памяти по адресу, вычисленному путем добавления $100 к содержимому регистра S, и последующего уменьшения значения регистра S на единицу.

Инструкции PLA и PLP извлекают значение из стека и помещают его в регистр A или регистр флагов, соответственно. При извлечении значения к содержимому регистра S сначала добавляется единица, после чего в целевой регистр передается значение

из ячейки памяти по адресу, вычисленному путем добавления $100 к содержимому регистра S.

Инструкция PLA обновляет состояние флагов N и Z. Инструкция PLP устанавливает или обнуляет шесть из семи флагов процессора в зависимости от значения, извлеченного из стека. Флаг B (флаг программного прерывания) имеет смысл только в копии регистра флагов процессора, которая была помещена в стек прерыванием или инструкцией PHP. Это отличает инструкцию BRK от запроса аппаратного прерывания. Обе инструкции, PHP и BRK, передают регистр флагов с установленным битом B (бит 4).

Аппаратные прерывания, генерируемые процессором на выводах $\overline{IRQ}$ (Interrupt Request — запрос на прерывание) и $\overline{NMI}$ (Non-Maskable Interrupt — немаскируемое прерывание), передают регистр флагов процессора с обнуленным битом B. Обработка прерываний и инструкция BRK будут рассмотрены позже в этой главе.

Арифметические инструкции

Как мы уже видели, сложение и вычитание выполняются с помощью инструкций ADC и SBC. Левым операндом каждой инструкции выступает содержимое регистра A, который также является местом назначения для результата операции. Для указания правого операнда доступны все режимы адресации. Значения флагов Z, C, N и V обновляются, отражая результат операции.

Инструкции INC и DEC, соответственно, увеличивают или уменьшают значение в указанной ячейке памяти на единицу. Результат сохраняется в той же ячейке памяти. Поддерживаются режимы абсолютной адресации и абсолютной индексной адресации. Эти инструкции обновляют состояние флагов N и Z в зависимости от результата операции.

Инструкции INX, DEX, INY и DEY увеличивают или уменьшают значение в регистре X или Y на единицу, согласно их названиям. Эти инструкции обновляют состояние флагов N и Z.

Инструкция CMP выполняет сравнение путем вычитания значения операнда из значения в регистре A. Поведение инструкции CMP очень похоже на последовательность, в которой за инструкцией SEC следует инструкция SBC. Значения флагов N, Z и C обновляются, отражая результат вычитания. Различия между инструкциями CMP и SBC состоят в следующем:

- CMP отбрасывает результат вычитания (выполнение инструкции CMP не влияет на значение в регистре A);
- CMP не использует десятичный режим, если установлен флаг D;
- CMP не влияет на состояние флага V;
- CMP поддерживает все режимы адресации.

Инструкции CPX и CPY похожи на CMP, за исключением того, что содержимое регистра X или Y используется в качестве левого операнда, согласно названию инструкции,

а для правого операнда поддерживаются только режимы абсолютной адресации и абсолютной индексной адресации.

Логические инструкции

Инструкции AND, EOR и ORA выполняют побитовые операции И, "исключающее ИЛИ" и ИЛИ, соответственно, между регистром A и операндом. Результат сохраняется в регистре A. Значения флагов Z и N обновляются, отражая результат операции. Поддерживаются все режимы адресации.

Инструкция ASL выполняет сдвиг операнда на один бит влево, вставляя 0 в младший бит. Старший бит сдвигается в флаг C. Это эквивалент умножения значения в регистре A на 2 и переноса старшего бита 9-битного результата в C.

Аналогично ASL инструкция LSR выполняет сдвиг операнда на один бит вправо, вставляя 0 в старший бит. Младший бит сдвигается в флаг C. Это эквивалентно делению беззнакового операнда на 2 со сдвигом остатка в C.

Инструкции ROL и ROR сдвигают содержимое регистра A на один бит влево или вправо соответственно. Предыдущее значение флага C переносится в позицию разряда, освободившегося в результате операции сдвига. Бит, выпадающий в результате сдвига из регистра A, сохраняется во флаге C.

Инструкции ASL, LSR, ROL и ROR поддерживают **режим адресации аккумулятора**, в котором в качестве операнда используется регистр A. Этот режим задается с помощью специального операнда со значением "A", как в инструкции ASL A. Эти четыре инструкции также поддерживают режимы абсолютной адресации и абсолютной индексной адресации.

Инструкция BIT выполняет побитовую операцию И между операндом и регистром A, результат при этом отбрасывается. Состояние флага Z обновляется в зависимости от результата операции. Биты 7 и 6 из ячейки памяти копируются во флаги N и V соответственно. Поддерживается только режим абсолютной адресации.

Инструкции ветвления

Инструкция JMP загружает операнд в ПС, и выполнение программы продолжается с инструкции в указанном месте. Адрес назначения, двухбайтовый абсолютный адрес, может находиться в любом месте адресного пространства процессора 6502:

- инструкции BCC и BCS выполняют условное ветвление, если флаг C обнулен или установлен, соответственно;
- инструкции BNE и BEQ выполняют условное ветвление, если флаг Z обнулен или установлен, соответственно;
- инструкции BPL и BMI выполняют условное ветвление, если флаг N обнулен или установлен, соответственно;
- инструкции BVC и BVS выполняют условное ветвление, если флаг V обнулен или установлен, соответственно.

Инструкции условного ветвления используют **режим относительной адресации**, где целевой адрес представляет собой 8-битное смещение со знаком (в диапазоне от –128 до +127 байт) от адреса инструкции, следующей за инструкцией ветвления.

Инструкции вызова подпрограммы и возврата из подпрограммы

Инструкция JSR передает в стек адрес инструкции, следующей за инструкцией JSR (минус один), загружает адрес, указанный в качестве 16-разрядного операнда, в ПС, а затем продолжает выполнение программы, начиная с инструкции в этом местоположении.

Инструкция RTS используется для завершения подпрограммы. Возвращаемое значение ПС (минус один) извлекается из стека и загружается ПС. Данная инструкция увеличивает значение ПС на единицу до того, как оно будет использовано в качестве адреса следующей команды для выполнения.

Инструкции для работы с флагами процессора

Инструкции для работы с флагами процессора предназначены для непосредственной установки или обнуления отдельных флагов.

- Инструкции SEC и CLC устанавливают и обнуляют флаг C, соответственно.
- Инструкции SED и CLD устанавливают и обнуляют флаг D, соответственно.
- Инструкция CLV обнуляет флаг V.

Ни одна из инструкций не устанавливает флаг V.

Инструкции для работы с прерываниями

Инструкции для работы с прерываниями позволяют процессору управлять обработкой внешних прерываний и генерировать собственные программные прерывания. Внешние прерывания могут быть двух типов: маскируемые и немаскируемые. Прерывания каждого типа запускаются с помощью выделенного для этой цели входного контакта на процессоре 6502.

Маскируемые прерывания можно отключить, установив флаг I процессора. При маскировании процессор игнорирует сигнал на соответствующем входном контакте. Немаскируемые прерывания, как следует из их названия, не могут быть заблокированы и всегда инициируют прерывание процессора, когда на соответствующем контакте обнаруживается соответствующий переход сигнала. Обработку прерываний мы рассмотрим более подробно в следующем разделе.

Инструкции SEI и CLI устанавливают и обнуляют флаг I, соответственно. Когда флаг I установлен, маскируемые прерывания отключены или маскированы.

Инструкция BRK генерирует немаскируемое прерывание. Адрес памяти через два байта после инструкции BRK помещается в стек, за ним следует регистр флагов про-

цессора. В ПС загружается адрес обработчика прерываний, который считывается по адресам $FFFE–$FFFF. Затем запускается выполнение обработчика прерываний.

Инструкция BRK не изменяет содержимое регистров (кроме указателя стека) или флагов процессора. Бит B регистра флагов, помещенного в стек, установлен, указывая на то, что прерывание является результатом выполнения инструкции BRK.

Инструкция RTI выполняет возврат из обработчика прерываний. Эта инструкция восстанавливает флаги процессора из стека, а также восстанавливает состояние ПС. После восстановления флагов процессора флаг B теряет свое значение, и его следует игнорировать.

Обработка прерываний и использование инструкции BRK будут обсуждаться в *разд. "Обработка прерываний" далее в этой главе.*

Инструкция отсутствия операций

Инструкция NOP (которую также часто называют *no-op*) ничего не делает, кроме перевода ПС к следующей инструкции.

Инструкции NOP иногда используются при разработке программ в качестве средства отладки. Например, одну или несколько инструкций можно эффективно "закомментировать", заполнив адреса памяти для этих инструкций байтами $EA. $EA — это шестнадцатеричное значение кода операции NOP в процессоре 6502.

Обработка прерываний

Процессоры обычно поддерживают некоторую форму обработки прерываний для реагирования на запросы обслуживания от внешних устройств. Концептуально обработка прерываний напоминает сценарий, в котором вы заняты решением некоторой задачи, и ваш телефон звонит. После ответа на вызов и, возможно, записи заметки для последующего действия (например, "купить хлеб") вы возвращаетесь к прерванной работе над задачей. Люди используют несколько подобных механизмов, таких как дверные звонки и будильники, которые позволяют нам прерывать менее приоритетные действия и реагировать на более насущные потребности.

Обработка $\overline{\text{IRQ}}$

Интегральная схема 6502 имеет два входа для сигналов, с помощью которых внешние компоненты уведомляют процессор о необходимости уделить внимание определенным событиям. Первый — это вход запроса на прерывание, $\overline{\text{IRQ}}$ (Interrupt ReQuest). Вход $\overline{\text{IRQ}}$ является инверсным, т. е. вызывает прерывание процессора при низком уровне (это означает, что сигнал считается активным при низком уровне или в состоянии 0, на что указывает линия над символами IRQ). Можно представить этот сигнал как телефонный звонок, уведомляющий процессор о входящем вызове.

Процессор 6502 не может мгновенно реагировать на низкий уровень сигнала на входе $\overline{IRQ}$. Прежде чем процессор 6502 сможет приступить к обработке прерывания, он должен завершить работу над уже выполняемой инструкцией. Затем он помещает в стек адрес возврата (адрес следующей инструкции, которая должна была быть выполнена после текущей инструкции), за которым следует регистр флагов процессора. Так как это прерывание было сгенерировано с помощью входа $\overline{IRQ}$, флаг B в строке флагов процессора, помещенной в стек, будет равен 0.

В отличие от инструкции `JSR`, адрес возврата, помещенный в стек в ответ на сигнал со входа $\overline{IRQ}$, является фактическим адресом следующей инструкции, которая должна быть выполнена, а не адресом инструкции за вычетом единицы. Адрес возврата после прерывания не будет увеличиваться на единицу, как это происходит при формировании адреса возврата во время выполнения инструкции `RTS`.

На следующем этапе обработки прерывания процессор загружает в ПС адрес программы обработки $\overline{IRQ}$, который считывается из памяти по адресам `$FFFE–$FFFF`. Затем процессор 6502 начинает выполнение кода обработчика прерывания по этому адресу. Обработчик прерывания — это код, который идентифицирует периферийное устройство, инициировавшее прерывание, и выполняет обработку, необходимую для удовлетворения запроса, а затем возвращает управление коду, который выполнялся до прерывания.

По завершении работы обработчика прерываний процессор выполняет инструкцию `RTI`. Она извлекает флаги процессора и состояние ПС из стека, после чего процессор возобновляет выполнение программы, начиная с инструкции, которая следовала за текущей инструкцией при переходе сигнала на входе $\overline{IRQ}$ на низкий уровень.

Вход $\overline{IRQ}$ генерирует **маскируемое прерывание**, означающее, что в этом случае можно выполнить действие, эквивалентное отключению звука при поступлении телефонного вызова. Когда начинается обработка прерывания $\overline{IRQ}$, процессор 6502 автоматически устанавливает флаг I, который маскирует (отключает) вход $\overline{IRQ}$ до снятия этого флага.

Флаг I обнуляется, когда инструкция `RTI` восстанавливает флаги процессора, поскольку этот флаг не мог быть установлен в момент реагирования процессора на сигнал $\overline{IRQ}$. Флаг I также можно снять с помощью инструкции `CLI` — это означает, что при обработке одного прерывания $\overline{IRQ}$ можно разрешить реагирование на другие прерывания $\overline{IRQ}$. Прерывание, обрабатываемое во время обработки другого прерывания, называется **вложенным прерыванием**.

Вход $\overline{IRQ}$ — это устройство, запускаемое по уровню сигнала. Это означает, что каждый раз, когда сигнал на входе $\overline{IRQ}$ переходит на низкий уровень и флаг I снят, процессор инициирует последовательность обработки прерывания. Одно из следствий этого заключается в том, что по завершении обработки прерывания взаимодействие процессора 6502 с источником прерывания должно гарантировать отсутствие низкого уровня сигнала на входе $\overline{IRQ}$. Если сигнал на входе $\overline{IRQ}$ останется низким на момент выполнения инструкции `RTI`, процессор 6502 сразу же начнет процесс обработки прерывания заново. Обработка прерываний, инициируемых со вхо-

да $\overline{IRQ}$, покрывает большинство рутинных операций взаимодействия между процессором 6502 и периферийными устройствами. Например, в большинстве компьютеров источником прерываний является клавиатура.

Каждое нажатие клавиши генерирует прерывание $\overline{IRQ}$. Во время обработки прерывания клавиатуры процессор 6502 считывает идентификатор нажатой клавиши через интерфейс клавиатуры и сохраняет его в очереди для последующей обработки активным в данный момент приложением. Коду обработчика прерываний $\overline{IRQ}$ не требуется информация о том, для чего будут использоваться данные о нажатии клавиши; он просто сохраняет их для последующего использования.

Обработка $\overline{NMI}$

Второй вход прерываний процессора 6502, $\overline{NMI}$ (Non-Maskable Interrupt), предназначен для **немаскируемых прерываний**. Как следует из его названия, вход $\overline{NMI}$ не маскируется с помощью флага I. $\overline{NMI}$ — это устройство, запускаемое по спаду сигнала.

Обработка прерываний $\overline{NMI}$ выполняется аналогично обработке прерываний $\overline{IRQ}$, за исключением того, что адрес процедуры обработки прерываний считывается из памяти по адресам `$FFFA–$FFFB`, а флаг I не оказывает влияния на прерывания этого типа.

Так как прерывание $\overline{NMI}$ является немаскируемым, оно может произойти в любое время, включая момент, когда процессор 6502 занят обработкой прерывания $\overline{IRQ}$ или даже обработкой произошедшего ранее прерывания $\overline{NMI}$.

Вход $\overline{NMI}$ обычно зарезервирован для событий с очень высоким приоритетом, которые нельзя отложить или пропустить. Один из возможных вариантов применения прерываний $\overline{NMI}$ — увеличение значения счетчика часов реального времени.

Приведенный ниже пример кода обработки $\overline{NMI}$ увеличивает значение 32-разрядного счетчика времени, размещенного по адресам `$10–$13`, каждый раз, когда происходит это прерывание:

```
; Увеличение значения 32-разрядного счетчика времени при каждом прерывании /NMI
NMI_HANDLER:
INC $10
BNE NMI_DONE
INC $11
BNE NMI_DONE
INC $12
BNE NMI_DONE
INC $13
NMI_DONE:
RTI
```

При обращении к аппаратным сигналам в исходном коде программы для обозначения сигнала с активным низким уровнем можно использовать в начале прямую косую черту. В комментарии к показанному выше примеру кода прерывание $\overline{\text{NMI}}$ обозначено как /NMI.

Обработка инструкции BRK

Инструкция BRK запускает обработку, которая очень похожа на обработку прерывания $\overline{\text{IRQ}}$. Так как BRK — это инструкция, перед началом обработки прерывания не требуется ждать завершения выполнения текущей инструкции. При выполнении инструкции BRK адрес возврата (адрес инструкции BRK плюс 2) и флаги процессора помещаются в стек, как при переходе сигнала на входе $\overline{\text{IRQ}}$ на низкий уровень. Обратите внимание, что при добавлении 2 к адресу инструкции BRK адрес возврата указывает не на первый, а на второй байт после инструкции BRK. Инструкция BRK является немаскируемой: состояние флага I не влияет на ее выполнение.

Обработчик инструкции BRK имеет тот же адрес, что и обработчик $\overline{\text{IRQ}}$, который находится в памяти по адресам $FFFE–$FFFF. Поскольку инструкция BRK и $\overline{\text{IRQ}}$ используют один и тот же обработчик, определить источник прерывания во время обработки можно по состоянию флага B. Флаг B в слове флагов процессора, помещенном в стек (это *не* флаг B в регистре флагов процессора P), устанавливается при выполнении инструкции BRK и обнуляется при обработке прерывания $\overline{\text{IRQ}}$.

Инструкция BRK отсутствует в большинстве приложений процессора 6502. Традиционно эта инструкция применяется для установки контрольных точек при отладке программы. Путем временной замены байта кода операции в месте, где требуется разместить контрольную точку, на инструкцию BRK можно передать контроль над выполнением программе отладки (часто называемой **монитором** в небольших компьютерных системах), благодаря чему пользователь сможет отображать и менять содержимое регистров и ячеек памяти перед возобновлением выполнения.

В следующем примере кода реализован минимальный обработчик, который различает прерывания $\overline{\text{IRQ}}$ и инструкции BRK. В нем в качестве места временного хранения используется адрес памяти $14:

```
; Обработка прерываний /IRQ и инструкций BRK
IRQ_BRK_HANDLER:
; Сохранение содержимого регистра A
STA $14
; Извлечение флагов процессора из стека в регистр A
PLA
PHA
; Проверка установки бита B в слове флагов в стеке
AND $10 ; $10 выбирает бит B
```

```
; Если результат отличен от нуля, то бит B был установлен:
; обработка для инструкции BRK
BNE BRK_INSTR
; бит B не был установлен: добавьте сюда обработку для прерывания /IRQ
; ...
JMP IRQ_DONE
BRK_INSTR:
; Добавьте сюда обработку для инструкции BRK
; ...
IRQ_DONE:
; Восстановление регистра A и возврат
LDA $14
RTI
```

Этот пример показал, как различать прерывания, инициируемые со входа $\overline{\text{IRQ}}$ процессора и инструкцией BRK, в архитектуре процессора 6502. В более сложных процессорах, включая те, которые мы обсудим в последующих главах, для каждого входного сигнала, инициирующего прерывание, предусмотрены уникальные векторы прерываний (начальные адреса обработчиков прерываний). Эти архитектуры также содержат обширную поддержку функций отладки, таких как установка контрольных точек в местах расположения указанных инструкций.

В предыдущих разделах были представлены категории инструкций в архитектуре процессора 6502 и дано краткое описание каждой инструкции в рамках этих категорий. Процессор 6502 намного проще, чем современные 32- и 64-разрядные процессоры, однако в этом обсуждении были представлены наиболее распространенные типы инструкций и режимов адресации, используемые даже в самых сложных современных процессорах, включая инструкции, поддерживающие универсальную концепцию обработки прерываний.

В следующем разделе будут представлены основы обработки ввода-вывода, которая обеспечивает передачу данных между процессором и периферийными устройствами.

Операции ввода-вывода

Целью части архитектуры процессора, отвечающей за операции ввода-вывода, является эффективная передача данных между внешними периферийными устройствами и памятью системы. Операции ввода переносят данные из внешнего мира в память, а операции вывода отправляют данные из памяти внешним адресатам.

Формат данных на внешней стороне интерфейса ввода-вывода может быть самым разным. Вот несколько примеров внешнего представления данных ввода-вывода компьютера:

- сигналы, передаваемые по видеокабелю, подсоединенному к монитору;

- колебания напряжения на проводах кабеля Ethernet;
- картина линий магнитного поля на поверхности диска;
- звуковые волны, генерируемые динамиками компьютера.

Независимо от того, какую форму данные принимают вне компьютера, соединение любого устройства ввода-вывода с процессором должно соответствовать архитектуре ввода-вывода процессора, а устройство ввода-вывода должно быть совместимо с любыми другими устройствами ввода-вывода. которые присутствуют в компьютерной системе.

Для взаимодействия с устройствами ввода-вывода процессор использует категории инструкций, режимы адресации и методы обработки прерываний, описанные ранее в этой главе. Отличие заключается в том, что вместо чтения и записи данных в системную память эти инструкции осуществляют считывание и запись данных в разделы памяти, которые используются для обмена данными с устройствами ввода-вывода.

В современных процессорах для доступа к устройствам ввода-вывода применяются два основных подхода: **ввод-вывод с распределением памяти** и **ввод-вывод с распределением по портам**. При вводе-выводе с распределением памяти устройствам ввода-вывода выделяются определенные части системного адресного пространства. Процессор обращается к периферийным устройствам по заранее определенным адресам с помощью тех же инструкций и режимов адресации, которые используются для чтения и записи в системную память. В процессоре 6502 для связи с периферийными устройствами принят подход с распределением памяти.

В процессорах, использующих ввод-вывод с распределением по портам, для выполнения операций ввода-вывода реализована отдельная категория инструкций. При этом для устройств ввода-вывода с распределением по портам выделяется адресное пространство, не относящееся к системной памяти. В качестве адресов устройствам ввода-вывода назначаются **номера портов**. В архитектуре *x*86 используется ввод-вывод с распределением по портам.

Одним из недостатков ввода-вывода с распределением памяти является необходимость выделения для устройств ввода-вывода части системного адресного пространства, что ведет к уменьшению максимального объема памяти, который может быть установлен в компьютерной системе. Недостаток ввода-вывода с распределением по портам — требование реализации в процессоре дополнительных инструкций для выполнения операций ввода-вывода.

В некоторых реализациях ввода-вывода с распределением по портам предусматривается предоставление дополнительных аппаратных сигналов, обозначающих обращение к устройству ввода-вывода, а не к системной памяти. Представив этот сигнал в виде селектора (который фактически становится еще одним битом адреса), одни и те же адресные строки можно использовать для доступа и к памяти, и к устройствам ввода-вывода. В качестве альтернативного решения для выполнения операций ввода-вывода с распределением по портам в некоторых более высокопроизводительных процессорах реализуют полностью выделенную шину. Такая архитектура позволяет одновременно выполнять операции ввода-вывода и доступа к памяти.

В простейшем подходе к вводу-выводу процессор сам обрабатывает все этапы операции ввода-вывода, используя инструкции для передачи данных между памятью и устройством ввода-вывода. В процессорах с более сложной архитектурой предусмотрены аппаратные функции для ускорения повторяющихся операций ввода-вывода. Мы обсудим три метода выполнения операций ввода-вывода с разной степенью использования процессора: программируемый ввод-вывод, ввод-вывод с управлением по прерываниям и прямой доступ к памяти.

Программируемый ввод-вывод

Программируемый ввод-вывод означает, что процессор выполняет каждый шаг операции передачи данных ввода-вывода с использованием программных инструкций. Рассмотрим клавиатуру, которая представлена процессору в виде двух отображенных в памяти однобайтовых адресов в адресной области ввода-вывода процессора. Один из этих байтов содержит информацию о состоянии, в частности бит, указывающий, когда была нажата клавиша. Второй байт содержит значение клавиши, которая была нажата.

При каждом нажатии клавиши устанавливается бит состояния *"нажата клавиша"*. При использовании программируемого ввода-вывода процессор должен периодически считывать регистр состояния клавиатуры, чтобы определить, была ли нажата клавиша. Если бит состояния указывает на нажатие клавиши, процессор считывает регистр данных клавиатуры, при этом бит состояния *"нажата клавиша"* обнуляется до следующего нажатия клавиши.

Если в регистре данных клавиатуры одновременно может храниться информация только об одной клавише, то операцию проверки состояния клавиатуры необходимо выполнять достаточно часто, чтобы ни одно нажатие клавиш не терялось, даже если за клавиатурой работает очень быстро печатающий пользователь. В результате процессор должен тратить значительное количество своего времени на проверку нажатий клавиш. При отсутствии быстрого ввода текста большинство из этих проверок будут безрезультатными.

Очевидно, что программируемый ввод-вывод не является высокоэффективным методом для решения задач общего назначения. По своей сути это похоже на проверку вашего телефона каждые несколько секунд, чтобы узнать, звонит ли вам кто-нибудь.

Программируемый ввод-вывод имеет смысл в отдельных ситуациях. Например, одноразовая настройка периферийного устройства в ходе запуска системы является вполне разумным применением этого метода.

Ввод-вывод с управлением по прерываниям

Устройство ввода-вывода может использовать прерывания для уведомления процессора о необходимости выполнения действий. В случае простого интерфейса клавиатуры вместо установки бита в регистре состояния периферийное устройство

может устанавливать на линии $\overline{\text{IRQ}}$ процессора 6502 низкий уровень для вызова прерывания при каждом нажатии клавиши. Такой подход позволяет освободить процессор от постоянных проверок нажатий клавиш, чтобы не отвлекать его от выполнения других операций. Процессор будет обращать внимание на интерфейс клавиатуры только тогда, когда необходимо выполнить работу, на что указывает прерывание. Использование прерываний для запуска операций ввода-вывода аналогично добавлению звонка в телефон, который мы должны были проверять на наличие входящих вызовов каждые несколько секунд при использовании программируемого ввода-вывода.

В процессоре 6502 для операций ввода-вывода предусмотрен единственный маскируемый **входной сигнал вызова прерываний** ($\overline{\text{IRQ}}$). Компьютерные системы обычно содержат несколько источников прерываний ввода-вывода. Это немного усложняет задачу обслуживания прерываний в 6502, поскольку процессор должен сначала определить, какое периферийное устройство инициировало прерывание, прежде чем он сможет начать передачу данных.

Для того чтобы определить источник прерывания, обработчик прерываний должен опросить каждое устройство, поддерживающее такие прерывания. В случае интерфейса клавиатуры эта операция опроса состоит из считывания регистра состояния клавиатуры, чтобы определить, установлен ли бит, указывающий на то, что произошло нажатие клавиши. Как только процессор идентифицирует устройство, ответственное за прерывание, он переходит к коду, который взаимодействует с устройством, для выполнения запрошенной задачи ввода-вывода. В случае интерфейса клавиатуры код обработки выполняет чтение регистра данных клавиатуры и очистку бита состояния, указывающего на нажатие клавиши, в результате чего входной сигнал $\overline{\text{IRQ}}$ отключается.

Прерывания от внешних устройств являются асинхронными событиями — это означает, что они могут произойти в любой момент времени. Проектирование компьютерной системы требует тщательного анализа даже самых маловероятных возможностей формирования прерываний, например во время запуска системы или при обработке других прерываний. Прерывания от нескольких устройств могут поступать одновременно или почти одновременно и в случайном порядке. Аппаратные схемы обработки прерываний и код обслуживания прерываний должны обеспечивать обнаружение и обработку всех прерываний независимо от их временны́х характеристик.

Управление вводом-выводом на основе прерываний избавляет процессор от необходимости периодически проверять устройства ввода-вывода, чтобы определить, требуется ли выполнение какого-либо действия. Однако обработка прерывания может занять значительное время процессора, если оно связано с передачей большого блока данных. Подобная ситуация часто возникает во время таких операций, как чтение с диска или запись на него. Следующий метод ввода-вывода, который мы обсудим, устраняет потребность в привлечении процессора к работе по передаче таких больших блоков данных.

Прямой доступ к памяти

Прямой доступ к памяти (direct memory access, DMA) позволяет периферийным устройствам получать доступ к системной памяти независимо от процессора. При использовании DMA для передачи блока данных процессор настраивает эту операцию, передавая в контроллер DMA начальный адрес перемещаемого блока данных, длину блока и адрес места назначения. После запуска процедуры DMA процессор может продолжить работу по выполнению других операций. После завершения операции контроллер DMA генерирует прерывание, чтобы сообщить процессору о том, что передача данных завершена.

В компьютерной системе контроллер DMA можно реализовать в виде отдельной интегральной схемы, управляемой процессором, либо архитектура процессора может содержать один или несколько встроенных DMA-контроллеров.

Устройства ввода-вывода, которые перемещают значительные объемы данных, такие как дисковые накопители, звуковые карты, видеокарты и сетевые интерфейсы, обычно используют DMA для эффективной передачи данных в системную память и из нее. Технология DMA также используется для передачи блоков данных в системной памяти.

Архитектура процессора 6502 не поддерживает операции DMA, но оригинальный IBM PC включал в себя DMA-контроллер. Архитектура почти каждого 32- или 64-разрядного процессора обеспечивает обширную поддержку операций DMA.

DMA — один из многих методов, которые повышают производительность компьютерной системы за счет ускорения повторяющихся операций. В *главах 5* и *9* мы рассмотрим несколько примеров использования DMA для ускорения операций ввода-вывода.

Резюме

В этой главе были описаны основные функциональные блоки простого процессора: блок управления, арифметико-логическое устройство и регистры. За описаниями этих компонентов последовал обзор инструкций и режимов адресации процессора. С целью демонстрации разнообразия и полезности инструкций, доступных в относительно простой процессорной архитектуре, были рассмотрены категории инструкций, реализованные в процессоре 6502.

Также в контексте архитектуры процессора 6502 были представлены и продемонстрированы концепции, связанные с обработкой прерываний. Глава завершилась обзором наиболее распространенных архитектурных подходов к управлению операциями ввода-вывода (ввод-вывод с распределением памяти и ввод-вывод с распределением по портам) и основных режимов выполнения операций ввода-вывода в компьютерной системе (программируемый ввод-вывод, ввод-вывод на основе прерываний и прямой доступ к памяти).

После завершения этой главы вы должны иметь общее представление о функциональных блоках процессора, обработке инструкций и прерываний, а также об опе-

рациях ввода-вывода. Эта информация является основой для следующей главы, в которой рассматривается архитектура на уровне компьютерной системы.

Упражнения

1. Рассмотрим сложение двух 8-битных чисел со знаком (т. е. чисел в диапазоне от –128 до +127), где один операнд положительный, а другой — отрицательный. Существует ли какая-либо пара 8-битных чисел с разными знаками, сумма которых выходит за пределы диапазона от –128 до +127? Такая ситуация будет представлять собой знаковое переполнение. Примечание: мы рассматриваем здесь только сложение, потому что, как мы видели, вычитание в архитектуре процессора 6502 — это то же самое, что и сложение, но с инверсией битов правого операнда.
2. Если ответ на *упражнение 1* — "нет", это означает, что единственный способ создать знаковое переполнение — сложить два числа с одинаковыми знаками. Если происходит переполнение, что вы можете сказать о результате выполнения операции "исключающее ИЛИ" между старшим битом каждого операнда и старшим битом результата? Другими словами, каков будет результат выражений `left(7) XOR result(7)` и `right(7) XOR result(7)`? В этих выражениях `(7)` указывает на бит 7 — старший бит.
3. Посмотрите на листинг VHDL в *разд. "Арифметико-логическое устройство" в этой главе* и определите, является ли логика установки и обнуления флага V корректной для операций сложения и вычитания. Проверьте результаты сложения 126 + 1, 127 + 1, –127 + (–1) и –128 + (–1). При пересылке данных через среду передачи, подверженную ошибкам, чтобы определить, были ли какие-либо биты данных потеряны или искажены во время передачи, обычно используется **контрольная сумма**. Она, как правило, добавляется к переданной записи данных. В одном из алгоритмов расчета контрольной суммы используются следующие шаги:

   1) сложение всех байтов в записи данных с сохранением только младших 8 бит суммы;

   2) определение контрольной суммы, которая представляет собой дополнение до двух этой 8-битной суммы;

   3) добавление байта контрольной суммы к записи данных.

   После получения блока данных с добавленной контрольной суммой процессор может определить правильность контрольной суммы простым сложением всех байтов в записи, включая контрольную сумму. Контрольная сумма является правильной, если младшие 8 бит суммы равны нулю. Реализуйте этот алгоритм контрольной суммы, используя язык ассемблера 6502. Байты данных начинаются с ячейки памяти по адресам `$10–$11`, а количество байтов (включая байт контрольной суммы) предоставляется в качестве входных данных в регистре X. Установите для регистра A значение 1, если контрольная сумма является правильной, и значение 0 в ином случае.

4. Поместите код проверки контрольной суммы из *упражнения 4* в помеченную подпрограмму, которую можно вызвать с помощью инструкции JSR и которая завершается инструкцией RTS.

5. Напишите и выполните набор тестов для проверки правильности работы подпрограммы проверки контрольной суммы, которую вы создали в *упражнениях 4–5*. Какова длина самого короткого блока данных, для которого ваш код может выполнить проверку контрольной суммы? Какова длина самого длинного блока данных?

4

Компоненты компьютерной системы

В этой главе представлены более низкоуровневые компоненты, используемые при построении компьютерных систем. Мы начнем с транзистора на основе структуры **металл-оксид-полупроводник** (МОП), который нашел широкое применение в схемах памяти, а также практически во всех других современных цифровых устройствах. Мы рассмотрим устройство компьютерной памяти на основе МОП-транзисторов и ее интерфейс с процессором. Мы познакомимся с современными компьютерными интерфейсами ввода-вывода, уделяя особое внимание высокоскоростной последовательной связи внутри компьютера, а также передаче данных через кабельные соединения внешним компонентам. Будут рассмотрены функциональные требования к устройствам системного ввода-вывода, включая дисплей, сетевой интерфейс, клавиатуру и мышь. Глава заканчивается наглядным примером технических характеристик материнской платы современного компьютера.

После прочтения этой главы вы получите четкое представление об аппаратных компонентах современных компьютерных систем — от их технических характеристик до схемного уровня. Вы будете знать, как реализована системная память, включая основы кеширования. Понимать механизмы эффективных операций ввода-вывода и использования интерфейса **Universal Serial Bus** (USB) для подключения клавиатуры, мыши и других устройств ввода-вывода. Вы получите представление о сетевом интерфейсе и познакомитесь с основными техническими характеристиками материнской платы современного компьютера.

В этой главе будут рассмотрены следующие темы:

- подсистема памяти;
- знакомство с полевыми МОП-транзисторами;
- построение схем динамической памяти с помощью полевых МОП-транзисторов;
- подсистема ввода-вывода;

* графические дисплеи;
* сетевой интерфейс;
* клавиатура и мышь;
* технические характеристики современных компьютерных систем.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Подсистема памяти

Подсистема памяти — это адресуемая последовательность ячеек памяти, содержащих инструкции и данные для использования процессором при выполнении программ. Современные компьютерные системы и цифровые устройства часто содержат более миллиарда 8-битных ячеек в основной памяти, каждая из которых может быть независимо считана и записана процессором.

Как было показано в *главе 1*, конструкция аналитической машины Бэббиджа в качестве средства хранения данных для вычислений включала в себя набор осей, на каждой из которых были установлены 40 колес, отражающих десятичные разряды. Считывание данных с оси было разрушительной операцией, приводящей к обнулению цифр на каждом из колес оси после завершения считывания. Это был полностью механический метод хранения данных.

С 1950-х по 1970-е годы предпочтительной технологией реализации цифровой компьютерной памяти был магнитный сердечник. Один бит памяти на магнитных сердечниках хранится в небольшом тороидальном (в форме пончика) керамическом постоянном магните. Сердечники набора, составляющего массив памяти, расположены в узлах прямоугольной сетки, образуемой горизонтальными и вертикальными соединительными проводами. Запись в битовую ячейку заключается в подаче по проводам, проходящим через сердечник, достаточного тока, чтобы можно было изменить полярность его магнитного поля. Бит 0 может быть определен как циркуляция магнитного потока внутри сердечника по часовой стрелке, а бит 1 — как циркуляция потока против часовой стрелки.

Считывание бита из памяти на магнитных сердечниках заключается в попытке установить для бита полярность 0 и наблюдении за электрическим откликом. Если выбранный сердечник уже содержит бит 0, то отклика не будет. Если сердечник содержит 1, то при изменении полярности возникает заметный импульс напряжения. Как и в аналитической машине Бэббиджа, операция чтения из памяти на магнитных сердечниках является разрушительной. После считывания битового значения 1 из памяти необходимо выполнить последующую запись, чтобы восстановить состояние бита.

Память на магнитных сердечниках является энергонезависимой: после отключения питания ее содержимое сохраняется в течение неопределенного срока. Она также обладает свойствами, которые делают ее ценной в таких областях применения, как космические аппараты, где важна устойчивость к радиации. Известно, что компьютеры космических кораблей системы "Спейс Шаттл" использовали память на магнитных сердечниках до конца 1980-х годов.

В современных компьютерных системах потребительского и бизнес-класса для основной системной памяти почти исключительно используются схемы динамической памяти произвольного доступа (dynamic random access memory, DRAM) на основе полевых МОП-транзисторов (metal-oxide-semiconductor field-effect transistor, MOSFET). В следующем разделе представлены особенности полевых МОП-транзисторов.

Знакомство с полевыми МОП-транзисторами

В *главе 2* описан *n-p-n*-транзистор, относящийся к типу **биполярных транзисторов** (bipolar junction transistor, BJT). Транзистор типа *n-p-n* называется биполярным, т. к. для его функционирования используются как положительные (*p*), так и отрицательные (*n*) носители заряда.

В полупроводниках носителями отрицательного заряда служат электроны. В работе полупроводника не участвуют физические частицы с положительным зарядом. Вместо этого при отсутствии обычно присутствующего электрона атом проявляет те же свойства, что и положительно заряженная частица. Эти недостающие электроны называются **дырками**. В биполярных транзисторах дырки выполняют функции носителей положительного заряда.

Концепция дырок настолько фундаментальна для работы полупроводников, что Уильям Шокли (William Shockley), один из изобретателей транзистора, написал книгу под названием "Электроны и дырки в полупроводниках" ("Electrons and Holes in Semiconductors"), которая была опубликована в 1950 г. Далее мы рассмотрим поведение положительных и отрицательных носителей заряда в униполярных транзисторах.

В отличие от биполярных транзисторов, **униполярный транзистор** полагается только на один из двух типов носителей заряда. **Полевой транзистор на основе структуры металл-оксид-полупроводник (полевой МОП-транзистор** — metal-oxide-semiconductor field-effect transistor, MOSFET) — это униполярный транзистор, пригодный для использования в качестве цифрового переключающего элемента. Как и транзистор типа *n-p-n*, полевой МОП-транзистор представляет собой трехполюсное устройство с управляющим входом для включения и выключения тока, протекающего через два других вывода. Выводы полевого МОП-транзистора называют **затвором**, **стоком** и **истоком**. Вывод затвора управляет протеканием тока между выводами стока и истока.

Полевые МОП-транзисторы относятся к категории устройств, работающих в **режиме обогащения носителями** или в **режиме обеднения носителями**. МОП-

транзистор в режиме обогащения носителями блокирует протекание тока между стоком и истоком, когда напряжение на затворе равно нулю, и разрешает протекание тока, когда напряжение на затворе превышает пороговое напряжение. МОП-транзистор в режиме обеднения носителями функционирует противоположным образом, блокируя протекание тока при высоком напряжении на затворе и разрешая его протекание, когда напряжение на затворе равно нулю.

На рис. 4.1 показано схематическое представление *n*-канального МОП-транзистора в режиме обогащения носителями.

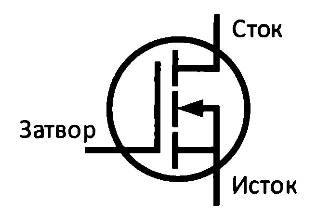


Рис. 4.1. *n-Канальный МОП-транзистор в режиме обогащения носителями*

Для наших целей *n*-канальный МОП-транзистор в режиме обогащения носителями функционирует как переключатель: когда напряжение на затворе относительно истока низкое (ниже порогового напряжения), между стоком и истоком возникает очень высокое сопротивление. Когда напряжение на затворе относительно истока высокое (выше порогового напряжения), сопротивление между этими выводами очень мало. Буква *n* в обозначении "*n*-канальный" связана с каналом в кремнии, который был легирован для увеличения числа электронов (носителей отрицательного заряда).

Поведение полевого МОП-транзистора напоминает работу *n-p-n*-транзистора, рассмотренного в *главе 2*. Однако есть важное отличие: МОП-транзистор — это устройство, управляемое напряжением, в то время как *n-p-n*-транзистор управляется током. При использовании *n-p-n*-транзистора в качестве переключателя, чтобы ток протекал между его эмиттером и коллектором, через базу должен протекать небольшой, но постоянный ток. В случае же МОП-транзистора для протекания тока между стоком и истоком требуется лишь приложить между затвором и истоком напряжение, величина которого выше порогового значения. Для удержания такого переключателя в открытом состоянии протекания тока через вход затвора практически не требуется. Благодаря этому МОП-транзистор потребляет значительно меньше энергии, чем *n-p-n*-транзистор, выполняющий эквивалентную цифровую функцию.

Мохаммед Аталла (Mohamed Atalla) и Давон Канг (Dawon Kahng) из Bell Telephone Laboratories изобрели МОП-транзистор в 1959 г. И только в начале 1970-х годов производственные процессы стали достаточно развитыми, чтобы обеспечить надежное изготовление интегральных схем на основе МОП-структуры. С тех пор МОП-транзистор является наиболее распространенным типом транзисторов, используемых в интегральных схемах. Согласно оценкам, в 2018 г. было изготовлено 13 секстиллионов транзисторов (секстиллион — это число, представляемое едини-

цей с 21 нулем), 99,9% из которых были МОП-транзисторами. МОП-транзистор является наиболее часто производимым устройством в истории человечества.

p-канальный МОП-транзистор в режиме обогащения носителями похож на *n*-канальный МОП-транзистор в режиме обогащения носителями, за исключением того, что он демонстрирует противоположную реакцию на напряжение между затвором и истоком: если напряжение на выводе затвора ниже, чем на выводе истока, и разница между ними превышает пороговое напряжение, то ток протекает между стоком и истоком, а если напряжение между затвором и истоком меньше этого порогового значения, то протекание тока между стоком и истоком блокируется. Буква *p* в обозначении "*p*-канальный" указывает на легирование канала, которое увеличивает количество дырок (носителей положительного заряда). На рис. 4.2 показано схематическое представление *p*-канального МОП-транзистора в режиме обогащения носителями.

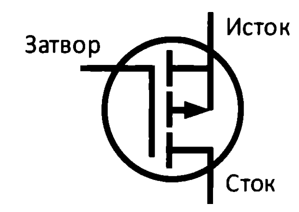


Рис. 4.2. *p-Канальный МОП-транзистор в режиме обогащения носителями*

Для того чтобы различать схематические представления *n*- и *p*-канальных МОП-транзисторов, обратите внимание, что в обеих конфигурациях вывод истока подсоединен к центральному из трех внутренних соединений. В *n*-канальном МОП-транзисторе стрелка на этом соединении указывает на затвор, а в *p*-канальном МОП-транзисторе — в направлении от затвора.

Как *n*-канальные, так и *p*-канальные МОП-транзисторы в режиме обогащения носителями можно считать нормально разомкнутыми переключателями. Это означает, что они не проводят ток, когда напряжение между затвором и истоком мало. Как *n*-канальные, так и *p*-канальные МОП-транзисторы также доступны в конфигурации, обеспечивающей работу в режиме обеднения носителями, что позволяет им функционировать как нормально замкнутые переключатели. Ток протекает через МОП-транзисторы, работающие в режиме обеднения носителями, когда напряжение между затвором и истоком мало, но не когда оно велико.

Для выполнения логических функций *n*- и *p*-канальный МОП-транзисторы часто используют в паре. Устройство, включающее в себя такую пару МОП-транзисторов, называется интегральной схемой на основе **комплементарной МОП-структуры** (КМОП-схемой). Название КМОП отражает тот факт, что транзисторы в каждой паре функционируют противоположным или взаимодополняющим образом. Вне периодов переключения КМОП-схемы почти не потребляют энергии, поскольку входы затворов не требуют практически никакого тока. Структуру КМОП-схемы разработали Чи-Тан Сах (Chih-Tang Sah) и Фрэнк Ванласс (Frank Wanlass) из компании Fairchild Semiconductor в 1963 г.

На рис. 4.3 показана схема вентиля НЕ, в которой *n-p-n*-транзистор из *главы 2* был заменен парой комплементарных полевых МОП-транзисторов.

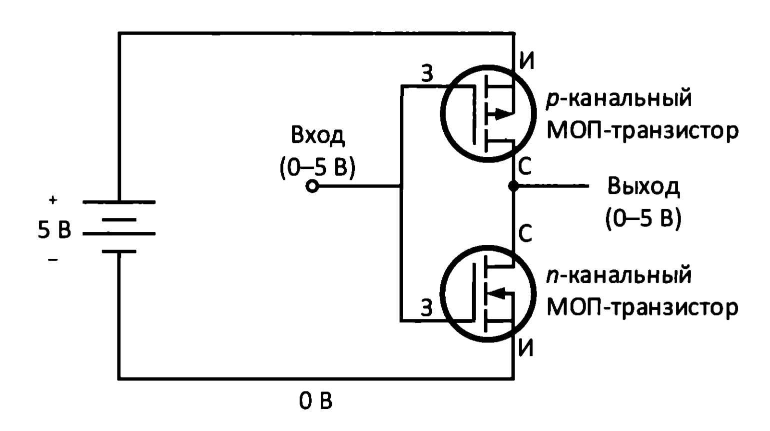


Рис. 4.3. *КМОП-схема вентиля НЕ*

Когда сигнал **Вход** имеет низкий уровень (вблизи 0 В), нижний *n*-канальный МОП-транзистор выключается, а верхний *р*-канальный МОП-транзистор включается. В результате **Выход** подключается к положительному полюсу источника напряжения, что вызывает увеличение уровня сигнала **Выход** примерно до 5 В. Когда сигнал **Вход** имеет высокий уровень, верхний МОП-транзистор выключен, а нижний МОП-транзистор включен, что вызывает снижение сигнала **Выход** почти до 0 В. Сигнал **Выход** всегда является инвертированным по отношению к сигналу **Вход**, что соответствует поведению вентиля НЕ.

Сегодня практически все цифровые интегральные схемы высокой плотности основаны на технологии КМОП. Помимо выполнения логических функций, полевой МОП-транзистор является ключевым компонентом современной архитектуры схем **оперативной памяти** (random access memory, RAM). В следующем разделе обсуждается использование полевых МОП-транзисторов в схемах памяти.

Построение схем динамической памяти с помощью полевых МОП-транзисторов

Один бит в стандартной интегральной схеме **динамической памяти произвольного доступа** (DRAM) состоит из двух схемных элементов: МОП-транзистора и конденсатора. В следующем разделе представлено краткое введение в электрические характеристики конденсаторов.

Конденсатор

Конденсатор — это пассивный схемный элемент с двумя выводами, способный накапливать энергию. Энергия поступает в конденсатор и покидает его в виде элек-

трического тока. Напряжение на выводах конденсатора пропорционально количеству содержащейся в нем электрической энергии.

Продолжая аналогию с гидравлической системой, описанной в *главе 2*, представьте, что конденсатор — это воздушный шарик, подсоединенный к боковой стороне трубы, ведущей к водопроводному крану. Давление воды в трубе заставляет шарик надуваться, наполняя его водой из трубы. Предположим, что это прочный воздушный шарик, и при надувании он растягивается, увеличивая внутреннее давление. Шарик наполняется до тех пор, пока давление в нем не сравняется с давлением в трубе, после чего перестает наполняться.

Если полностью открыть кран на конце трубы, то в результате выпуска воды давление в трубе уменьшится. Вода из шарика будет вытекать обратно в трубу до тех пор, пока давление в шарике снова не сравняется с давлением в трубе.

Гидравлические устройства, называемые **амортизаторами гидравлических ударов**, функционируют именно таким образом, решая проблему труб, которые издают стучащие звуки при открывании и закрывании водопроводных кранов. Амортизатор гидравлических ударов демонстрирует поведение, подобное растяжению воздушного шарика, сглаживая резкие изменения давления воды, возникающие при открывании и закрывании кранов.

Количество электрической энергии, содержащейся в конденсаторе, аналогично количеству воды в воздушном шарике. Напряжение на конденсаторе аналогично давлению внутри воздушного шарика, возникающему при его растяжении.

Электрический конденсатор можно изготовить из двух параллельных металлических пластин, разделенных изолирующим материалом, например воздухом. Выводы конденсатора подсоединены к каждой из пластин. Отношение количества запасенной электрической энергии к напряжению на конденсаторе называется **емкостью**. Она зависит от размера параллельных пластин, расстояния между ними и типа материала, используемого в качестве изолятора. Емкость конденсатора аналогична размеру воздушного шарика в примере с гидравлической системой. Конденсатор с большей емкостью соответствует шарику большего размера. Для наполнения большого воздушного шарика до заданного давления требуется больше воды, чем для наполнения маленького.

Схематическое обозначение конденсатора показано на рис. 4.4.



Рис. 4.4. *Схематическое обозначение конденсатора*

Две горизонтальные линии с зазором между ними отражают конструкцию конденсатора с металлическими пластинами, описанную в этом разделе. Единицей измерения емкости является **фарад**, названный в честь английского ученого Майкла Фарадея, который, помимо многих других своих достижений, изобрел электродвигатель.

Битовая ячейка динамической памяти

Битовая ячейка динамической памяти (dynamic random access memory, DRAM) — это доступное для чтения и записи место хранения одного бита данных. Модуль памяти DRAM в современном компьютере или телефоне содержит миллиарды битовых ячеек. Один бит в схеме DRAM состоит из МОП-транзистора и конденсатора, расположенных так, как показано на рис. 4.5.

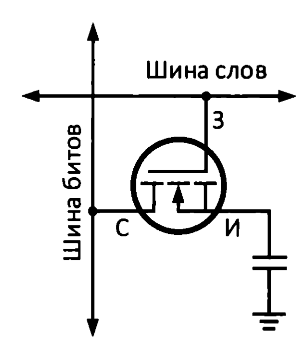


Рис. 4.5. *Схема битовой ячейки DRAM*

На этом рисунке символ с тремя горизонтальными линиями в правом нижнем углу является символом **земли** — стандартного представления опорного напряжения 0 В, которое мы использовали в предыдущих схемах, таких как показанная на рис. 4.3. **Шина слов** и **шина битов** — это провода, используемые для соединения отдельных битовых ячеек в сетку.

Эта однобитовая ячейка повторяется в прямоугольной сетке, образуя полный банк памяти DRAM. На рис. 4.6 показана конфигурация 16-битного банка памяти DRAM, состоящего из 4 слов по 4 бита каждое.

Реальные банки памяти DRAM содержат гораздо большее количество битов, чем показанная здесь простая схема. Типичные устройства DRAM имеют размер слова 8 бит, а не 4 бита, обозначенные на этом рисунке как B_0–B_3. Это означает, что схема DRAM позволяет параллельно сохранять или извлекать 8 бит.

Количество экземпляров битовых ячеек на шине слов в реальном массиве банков DRAM является целым числом, кратным размеру слова устройства. Большие модули DRAM, используемые в персональных компьютерах, содержат много слов на каждой шине слов. Например, микросхема DRAM с 8-битовыми словами и 1024 словами в строке содержит 8192 бита в строке, при этом все выводы затворов МОП-транзисторов вдоль строки управляются одним сигналом по шине слов. Эти устройства содержат дополнительную логику мультиплексора для выбора конкретного запрашиваемого процессором слова из множества слов в ряду, выбранном активной шиной слов.

Вертикальное измерение банка памяти DRAM состоит из реплицированных копий рядов ячеек с одной шиной слов, управляющей каждым рядом. Шина слов соеди-

няет все битовые ячейки по горизонтали, а сигналы шины битов — ячейки во всех строках по вертикали.

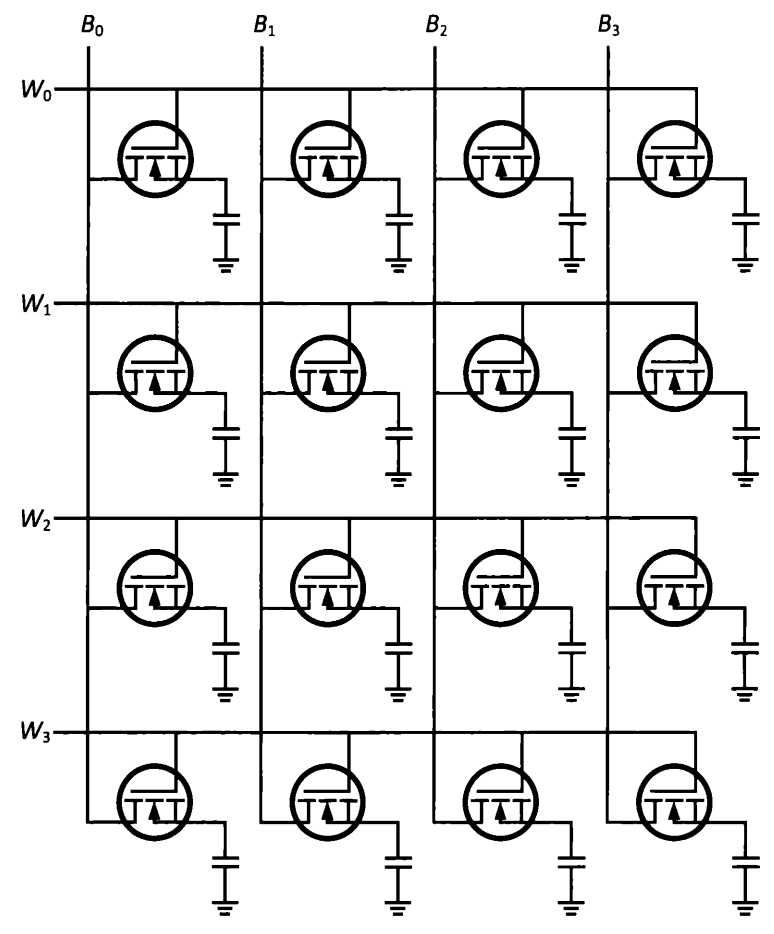


Рис. 4.6. *Организация банка памяти DRAM*

Состояние каждого бита памяти хранится в конденсаторе ячейки. Низкое напряжение на конденсаторе отражает значение 0, а высокое напряжение — значение 1. В контексте устройств памяти DRAM DDR5 низкое напряжение — около 0 В, а высокое — около 1,1 В.

На шине слов для каждого ряда бóльшую часть времени поддерживается низкое напряжение. Это обеспечивает нахождение МОП-транзисторов в выключенном состоянии, при котором сохраняется состояние конденсатора. Когда приходит время прочитать слово (фактически — целый ряд) из устройства DRAM, схема адресации выбирает соответствующую шину слов и подает на нее напряжение высокого уровня, а на всех остальных шинах слов в банке сохраняется низкое напряжение. При этом МОП-транзисторы в каждой битовой ячейке на активной шине слов открываются, подавая напряжение конденсаторов этих ячеек на подключенные шины битов. Шины битов для ячеек в состоянии 1 (высокий уровень) будут иметь более вы-

сокое напряжение, чем шины для ячеек в состоянии 0 (низкий уровень). Напряжения на шинах битов считываются схемой устройства DRAM и фиксируются в выходном регистре микросхемы.

Запись в слово DRAM начинается с установки высокого уровня на выбранной шине слов таким же образом, как и при чтении слова. Вместо измерения напряжений на шинах битов устройство DRAM подает на каждую битовую линию напряжение, которое должно быть записано в каждую ячейку — 0 или 1,1 В, в зависимости от того, какое значение должен получить бит данных — 0 или 1. Как и при наполнении или опорожнении воздушного шарика, каждому конденсатору требуется некоторое время, чтобы зарядиться или разрядиться до напряжения, действующего на шине битов, к которой он подсоединен. По истечении этой задержки на шину слов подается низкий уровень, чтобы закрыть МОП-транзисторы и зафиксировать новые уровни заряда конденсаторов.

Технология схем DRAM осложняется тем фактом, что конденсаторам свойственна утечка. После зарядки конденсатора до ненулевого напряжения заряд со временем утекает, уменьшая напряжение на конденсаторе. Из-за этого содержимое каждой ячейки необходимо периодически обновлять.

Операция обновления считывает значение каждой ячейки и затем записывает его обратно в ячейку. При этом конденсатор перезаряжается до "полного" уровня напряжения, если он представляет значение 1, или разряжается почти до 0 В, если он представляет значение 0. Типичный интервал обновления для современных устройств DRAM составляет 64 миллисекунды. Обновление DRAM происходит непрерывно, строка за строкой, в фоновом режиме во время работы системы и синхронизируется с доступом процессора к памяти, чтобы избежать конфликтов. В ходе этого процесса наблюдается небольшое снижение производительности, которое возникает, когда доступ процессора к памяти задерживается из-за выполнения обновления.

Необходимость периодического обновления значительно усложняет конструкцию систем, использующих устройства памяти DRAM, однако преимущества хранения состояния бита с помощью всего лишь одного транзистора и одного конденсатора настолько велики, что DRAM вытеснила все альтернативы в качестве предпочтительной технологии для основной памяти в компьютерных системах потребительского, коммерческого и научного назначения.

В следующем разделе будет рассмотрена архитектура текущего поколения технологии DRAM — DDR5.

SDRAM DDR5

Компания Intel выпустила первую коммерческую интегральную схему DRAM в 1970 г. Intel 1103 содержала 1024 бита и имела размер слова 1 бит. Обновление в Intel 1103 необходимо было выполнять каждые 2 миллисекунды. В начале 1970-х годов память DRAM на МОП-устройствах опередила память на магнитных сердечниках в качестве предпочтительной технологии памяти в компьютерных системах.

DRAM является энергозависимой технологией, т. е. при отключении питания заряд в конденсаторах битовых ячеек утекает, и данные теряются.

Термин **"удвоенная скорость передачи данных"** (double data rate, DDR) относится к временны́м характеристикам обмена данными между модулем памяти и контроллером памяти процессора. Исходная технология DRAM с **базовой скоростью передачи данных** (single data rate, SDR) предусматривала выполнение одной операции передачи данных за цикл тактового сигнала памяти. Устройства памяти DDR выполняют две операции передачи за цикл тактового сигнала: одну — по фронту тактового сигнала, и вторую — по его спаду. Цифра после "DDR" обозначает поколение технологии DDR. Таким образом, DDR5 — это пятое поколение стандарта DDR. Термин **"синхронная DRAM"** (synchronous DRAM, SDRAM) указывает на то, что схема DRAM синхронизирована с контроллером памяти процессора с помощью общего тактового сигнала. Текущим поколением технологии DRAM, широко используемой в настоящее время, является SDRAM DDR4, а SDRAM DDR5 только начинает выходить на рынок.

Современные персональные компьютеры и мобильные устройства, такие как смартфоны, как правило, содержат несколько **гигабайтов (Гбайт)** оперативной памяти. 1 Гбайт содержит 2^{30} байт, что эквивалентно 1 073 741 824 (чуть более одного миллиарда) байтам. Как следует из названия, память произвольного доступа позволяет процессору за одну операцию считывать или записывать любую ячейку памяти в адресном пространстве оперативной памяти. По состоянию на 2021 г. высококлассный модуль памяти для использования в ноутбуках содержит 32 Гбайт памяти DRAM, распределенной между 16 интегральными схемами. Каждая схема DRAM в этом модуле содержит 2 гигаслова (1 гигаслово — это 2^{30} слов) при длине одного слова 8 бит.

В 2021 г. ведущим стандартом модулей памяти должен был стать SDRAM DDR5 — усовершенствованная и оптимизированная технология DRAM, основанная на поколениях DDR1, DDR2, DDR3 и DDR4. Модуль памяти DDR5 помещен в корпус типа **DIMM** (dual inline memory module — **модуль памяти с двухрядным расположением выводов**). Для подсоединения к разъему DIMM на материнской плате модуль DIMM оснащен электрическими контактами по обе стороны печатной платы (отсюда термин *"двухрядный"* в его названии). Стандартный разъем DIMM DDR5 имеет 288 контактов. Для таких систем, как ноутбуки, где свободное пространство ограничено, предусмотрен более компактный формат, называемый **SODIMM** (small outline DIMM). Стандартный разъем SODIMM DDR5 имеет 262 контакта. Из-за уменьшенного количества контактов модули SODIMM лишены функций, которые поддерживают некоторые модули DIMM, например способности обнаруживать и исправлять битовые ошибки в данных, извлекаемых из устройства. Эту способность обеспечивает применение **кода коррекции ошибок** (error correcting code, ECC).

Модули памяти DDR5 обычно рассчитаны на напряжение питания 1,1 В. Показательный пример — отдельный модуль DDR5 может выполнять до 4,8 млрд операций передачи данных в секунду, что вдвое превышает тактовую частоту памяти 2400 МГц. При скорости передачи 8 байт/с это устройство DDR5 теоретически мо-

жет передавать в секунду 38,4 Гбайт данных. Модули DDR5 будут доступны с различными тактовыми частотами, объемами памяти и по разным ценам.

В реальных модулях DRAM используются прямоугольные банки однобитных ячеек, которые были описаны в предыдущем разделе, однако внутренняя архитектура устройств DDR5 несколько сложнее. Интегральная схема DRAM обычно содержит несколько банков. Перед выполнением операции чтения или записи логика адресации выбирает банк, содержащий нужную ячейку памяти. В модулях DDR5 банки дополнительно организованы **в группы банков**, что требует дополнительной логики адресации для выбора нужной группы. Устройство DDR5 содержит до 8 групп банков, каждая из которых содержит до 4 банков. Причина разделения архитектуры модуля DDR5 на несколько групп банков заключается в том, чтобы довести скорость передачи данных до максимума за счет параллельного выполнения нескольких одновременных операций доступа к памяти с перекрытием. Это позволяет передавать данные между процессором и оперативной памятью с максимальной скоростью, сводя к минимуму потребность в ожидании завершения каждой операции доступа к DRAM.

Помимо указания правильного адреса в модуле памяти DDR5 система должна передать посредством интерфейсных сигналов команду, указывающую действие, которое должно быть выполнено: чтение из памяти, запись в память или обновление состояния выбранной строки.

В стандарте DDR5 SDRAM, который можно приобрести в **Объединенном совете по проектированию электронных устройств** (Joint Electron Device Engineering Council, JEDEC) по адресу **https://www.jedec.org/standards-documents/docs/jesd79-5**, приведено подробное определение интерфейса памяти DDR5 для компьютерных систем. Этот стандарт содержит всю информацию, необходимую для разработки модулей памяти, совместимых с любой компьютерной системой, поддерживающей DDR5.

Исторически сложилось так, что каждое новое поколение стандартов DDR SDRAM было несовместимо с предыдущими поколениями. Материнская плата, созданная для модулей памяти DDR5, будет работать только с модулями DDR5. Разъем для каждого поколения модулей DDR устроен таким образом, что вставить в него модуль другого поколения невозможно. Например, модуль DRAM DDR4 несовместим с разъемом DDR5.

По мере развития технологий памяти основными улучшениями в каждом новом поколении являются увеличение скорости передачи данных и повышение плотности памяти. Для достижения этих целей в более поздних поколениях было уменьшено напряжение питания, что позволило снизить энергопотребление системы и создать более плотные схемы памяти, избегая при этом чрезмерного нагрева.

Большинство современных процессоров воспринимают системную память как линейный массив последовательных адресов. Процессоры с менее изощренной архитектурой, такие как 6502, напрямую обращаются к микросхемам оперативной памяти, используя адреса памяти, указываемые в инструкциях. Из-за сложности управляющих сигналов и логики управления банками памяти в устройствах

SDRAM DDR5 современные компьютерные системы оснащают контроллером памяти, который преобразует каждый линейный адрес процессора в сигналы управления и адресации, выбирающие соответствующий модуль DDR5 (в системе с несколькими модулями памяти), группу банков, банк и расположение строк/столбцов в выбранном банке. Контроллер памяти — это последовательностное логическое устройство, управляющее обменом данными между процессором и модулями оперативной памяти DRAM. Для достижения максимальной производительности системы контроллер памяти должен интеллектуально использовать возможности перекрытия операций, предоставляемые модулями памяти DDR5.

Сложные современные процессоры обычно интегрируют функцию контроллера памяти в саму интегральную схему процессора. Также можно разработать систему с отдельным контроллером памяти, который располагается между процессором и оперативной памятью.

Интерфейс контроллера памяти может содержать несколько каналов, где каждый канал представляет собой отдельный путь обмена данными между процессором и одним или несколькими модулями памяти. Преимущество выделения нескольких каналов в архитектуре памяти заключается в том, что такая конфигурация позволяет реализовать одновременный доступ к памяти по этим каналам. Однако система, содержащая несколько каналов доступа к памяти, не обеспечивает автоматического увеличения скорости доступа к памяти. Системное программное обеспечение должно активно управлять назначением областей памяти для каждого приложения или системного процесса, чтобы сбалансировать использование памяти между каналами. Если бы операционная система просто последовательно распределяла процессы по областям физической памяти, сначала заполняя один модуль памяти, а затем переходя к следующему, то многоканальный доступ к памяти не принес бы никакой пользы, поскольку все процессы были бы вынуждены использовать один и тот же канал памяти.

DDR для графики

DDR для графики (graphics DDR, GDDR) — это технология памяти DDR, оптимизированная для использования в качестве оперативной видеопамяти в графических контроллерах. GDDR имеет более широкую шину памяти для поддержки предъявляемых средствами отображения требований высокой пропускной способности. Стандартная память DDR, с другой стороны, оптимизирована для обеспечения минимальной задержки доступа к данным.

Номера поколений технологий GDDR и DDR не согласованы друг с другом. По состоянию на 2021 г. модули GDDR6 доступны уже несколько лет, в то время как стандарт SDRAM DDR6 по-прежнему находится в разработке.

Предварительная выборка

Одним из атрибутов производительности DRAM, который весьма незначительно улучшается от одного поколения к другому, является скорость чтения или записи

конкретного бита. Для того чтобы добиться увеличения средней скорости передачи данных в модули DRAM и из них, устройства должны использовать другие способы повышения производительности. Одним из методов достижения более высокой средней скорости передачи данных является **предварительная выборка**.

Идея **предварительной выборки** заключается в использовании того факта, что всякий раз, когда процессор обращается к определенной ячейке памяти, появляется вероятность, что вскоре он обратится к адресам, близким к этой исходной ячейке. Идея предварительной выборки заключается в чтении большего блока памяти, чем один адрес, на который ссылается инструкция процессора, и передаче всего блока из устройства DRAM в процессор. В контексте модуля памяти DDR5 размер блока обычно составляет 64 байта.

Модуль DDR5 может быстро прочитать 64 байта, т. к. он обращается ко всем 512 битам этих 64 байтов одновременно. Для этого модуль DDR5 считывает целое число, кратное 512 шинам битов, из ячеек, выбранных шиной слов. Биты выбранного ряда считываются одновременно, затем проходят через мультиплексор для извлечения нужных 512 бит из (возможно) 8192 бит всего ряда, которые затем фиксируются в выходном регистре. Зафиксированные биты передаются из модуля DRAM в процессор по тактовому сигналу DDR.

При эффективном использовании нескольких групп банков множественные операции чтения памяти и передачи полученных данных могут перекрываться по времени и обеспечивать перемещение данных между модулем памяти и процессором с максимальной скоростью, которую может поддерживать интерфейс.

Получив блок длиной 64 байта, процессор сохраняет эти данные во внутренней кеш-памяти и выбирает конкретный элемент данных (возможно, размером всего 1 байт) из блока, запрашиваемого выполняемой инструкцией. Если последующая инструкция обращается к другим данным, содержащимся в том же блоке, процессору достаточно обратиться к локальному кешу, что приводит к гораздо более быстрому выполнению этой операции, чем с помощью инструкции, которая первоначально извлекала этот блок данных из модуля DRAM.

Помимо взаимодействия с основной памятью, процессор должен обмениваться данными с внешним миром через устройства ввода и вывода. В следующем разделе рассматривается реализация интерфейсов ввода-вывода в современных компьютерных системах.

Подсистема ввода-вывода

В *главе 3* были представлены две крупные категории архитектур ввода-вывода: ввод-вывод с распределением памяти и ввод-вывод с распределением по портам. Плюсы и минусы каждого из этих подходов играли существенную роль на заре развития персональных компьютеров, когда количество физических адресных линий ограничивало общее пространство памяти процессора диапазоном в 1 Мбайт. Архитектуры современных процессоров могут адресовать гораздо больший диапазон памяти, обычно в десятки гигабайт. Следствием такого расширения адресного про-

странства является доступность адресных областей для использования в интерфейсах ввода-вывода. Благодаря этому современные 32- и 64-разрядные процессоры общего назначения используют ввод-вывод с распределением памяти для большинства своих интерфейсов.

Сложные современные процессоры обычно реализуют контроллер памяти на микросхеме процессора, напрямую обмениваясь данными с модулями памяти DDR. Большинство других типов операций ввода-вывода, выполняемых этими процессорами, передаются одной или нескольким внешним интегральным схемам, которые обычно называют **микропроцессорным набором** или **чипсетом**. Термин "чипсет", как правило, используется даже в тех случаях, когда для реализации функций ввода-вывода требуется всего одна микросхема.

Чипсет предоставляет интерфейсы для широкого спектра периферийных устройств, включая дисковые накопители, сетевые адаптеры, клавиатуры, мыши и многие другие устройства с поддержкой USB. Большинство этих интерфейсов реализованы на основе того или иного вида последовательной шины. В следующих разделах представлены наиболее распространенные технологии ввода-вывода, используемые в современных компьютерах.

Параллельные и последовательные шины данных

Параллельная шина данных одновременно передает несколько битов данных по отдельным проводникам между двумя или бóльшим числом конечных точек. В ранних ПК параллельные шины использовались для таких функций, как подключение принтера к компьютеру. Со временем стали очевидны некоторые ограничения параллельных шин.

- В зависимости от разрядности для подключения параллельной шины может потребоваться много проводов. Это означает, что кабели стоят дороже и существует повышенная вероятность возникновения проблем из-за обрыва проводов кабеля или ненадежного электрического контакта в разъемах.
- По мере того, как разработчики компьютерных систем пытались увеличить скорость передачи данных по шине (и тем самым получить конкурентное преимущество), существенную роль стало играть еще одно ограничение параллельных шин: несмотря на то, что устройство, передающее слово данных по шине, способно выводить все его биты одновременно, отдельные сигналы могут прибывать в точку назначения в разное время.

  Это может быть вызвано различиями в эффективной длине пути в проводниках кабеля или печатной платы. Этим обстоятельством обусловлено существование верхнего предела скорости передачи данных, которую может поддерживать архитектура параллельной шины.

Еще одно ограничение параллельных шин — они могут передавать данные единовременно только в одном направлении (так называемый **полудуплексный режим**), если не предусмотрен дублирующий набор соединений для одновременной передачи данных в противоположном направлении. По этой причине параллельные шины

обычно не обеспечивают одновременную двустороннюю связь, которую называют **полнодуплексным режимом**.

Последовательная шина данных передает данные между двумя конечными точками по одному биту за раз с помощью пары проводников. Большинство высокоскоростных каналов связи между процессором и периферийными устройствами в современных компьютерах используют тот или иной вариант последовательной шины. На первый взгляд, переход от параллельной шины к последовательной влечет за собой существенную потерю пропускной способности, однако последовательные шины обладают рядом важных преимуществ, которые делают их применение привлекательным при решении задач, предъявляющих высокие требования к производительности.

Высокоскоростные последовательные шины в персональных и коммерческих компьютерах передают данные по парам проводников с использованием **дифференциальной передачи сигналов**. В этом случае используются два проводника, тщательно подобранных по длине и обладающих почти одинаковыми электрическими характеристиками. В кабелях эти проводники представляют собой изолированные провода, скрученные друг с другом для образования **витых пар**.

На рис. 4.7 представлена последовательная шина данных с дифференциальной передачей сигналов.

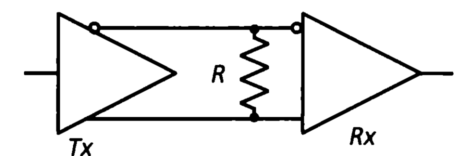


Рис. 4.7. *Схема последовательной шины с дифференциальной передачей сигналов*

Цифровой сигнал, который необходимо передать, поступает в передатчик (обозначенный **Tx**) через вход в левой части рисунка. Входной сигнал преобразуется в напряжения на двух параллельных линиях, показанных в средней части схемы. Маленький кружок указывает на то, что сигнал, идущий от передатчика по верхней линии, инвертирован относительно сигнала в нижней линии.

В типичном последовательном интерфейсе высокий уровень сигнала на входе передатчика будет генерировать напряжение 1,0 В на верхнем проводнике последовательной шины и 1,4 В на нижнем проводнике. Входной сигнал низкого уровня приведет к появлению напряжения 1,4 В на верхнем проводнике и 1,0 В на нижнем проводнике.

Входы приемника (обозначенного **Rx**) имеют высокое сопротивление — это означает, что приемник потребляет незначительный ток из цепи. Приемник измеряет напряжение на резисторе, типичное сопротивление которого составляет 100 Ом. Когда на входе **Tx** высокий уровень, верхний вывод резистора находится под напряжением –0,4 В относительно нижнего вывода. Когда на входе **Tx** низкий уровень, верхний вывод резистора находится под напряжением +0,4 В относительно нижнего вывода.

Приемник генерирует свой выходной сигнал, инвертируя один из входов (верхний на рис. 4.7, с маленьким кружком) и добавляя полученное напряжение к напряжению на другом входе. Другими словами, приемник измеряет только разницу между напряжениями на двух проводниках. Фундаментальное преимущество этого подхода заключается в том, что большинство искажающих сигналы помех вызывает колебания напряжения в проводниках, несущих сигнал. При размещении двух проводников очень близко друг к другу бóльшая часть шумового напряжения, вносимого в один из проводников, также будет проявляться и в другом проводнике. Операция вычитания устраняет большýю часть шума, который в противном случае мешал бы точному обнаружению сигнала приемником.

Последовательная шина данных может передавать несколько миллиардов битов в секунду, что намного больше, чем параллельная шина в старых ПК. Кроме того, можно использовать несколько последовательных шин рядом друг с другом, что позволяет умножить пропускную способность в зависимости от количества шин.

Принципиальное различие между несколькими последовательными шинами, соединяющими две конечные точки, и параллельной шиной, реализующей такое же соединение, состоит в том, что во многих стандартах интерфейсов последовательные шины работают до определенной степени независимо. Для них не требуется синхронизация на уровне передачи каждого бита, как в случае передачи по параллельной шине. Это упрощает устройство соединений, способных поддерживать очень высокие скорости передачи данных, при этом нужно заботиться лишь о точном соответствии длины проводника и электрических характеристик в каждой паре последовательных проводников.

Связь между современным процессором и чипсетом его материнской платы обычно осуществляется по нескольким последовательным шинам данных, называемым **линиями высокоскоростного ввода-вывода** (high-speed input output, HSIO). Каждая линия представляет собой последовательное соединение с одним каналом передачи данных в каждом направлении (например таким, как на рис. 4.7), благодаря чему обмен данными осуществляется в полнодуплексном режиме.

Отдельные линии HSIO могут быть назначены определенным типам периферийных интерфейсов, реализованных в виде последовательных соединений, таких как PCI Express, SATA, M.2, USB и Thunderbolt. В следующих разделах представлен каждый из этих стандартов.

PCI Express

Оригинальная шина **Peripheral Component Interconnect** (**PCI**) представляла собой 32-разрядную параллельную шину, работающую на частоте 33 МГц, которая использовалась в IBM PC-совместимых компьютерах примерно с 1995 по 2005 г.

В разъемы PCI на материнских платах компьютеров можно было установить множество карт расширения с разными функциями, включая сетевой интерфейс, адаптер дисплея или плату звукового выхода. К началу 2000-х годов ограничения архи-

тектуры параллельной шины стали сдерживать дальнейшее развитие, и началась разработка замены PCI в виде последовательной шины под названием PCI Express.

PCI Express (PCIe) представляет собой двунаправленную последовательную шину с дифференциальной передачей сигналов, используемую в основном для соединения конечных точек обмена данными на материнских платах компьютеров. Пропускная способность PCIe выражается в миллиардах пересылок данных (гигатранзакций) в секунду или ГТ/с. Одна транзакция — это передача одного бита по шине от передатчика к приемнику. В каждый многобитный пакет PCIe добавляет дополнительные избыточные биты для контроля целостности данных. Разные поколения PCIe используют различное количество этих служебных битов, влияющих на фактическую скорость передачи данных. В табл. 4.1 приведены данные основных поколений PCIe с указанием года их появления, скорости передачи по одной линии в гигатранзакциях в секунду (ГТ/с) и эффективной скорости передачи данных в мегабитах в секунду (Мбит/с).

Таблица 4.1. *Поколения шины PCI Express*

Поколение PCIe	Год появления	Скорость передачи, ГТ/с	Эффективная скорость передачи данных в одну сторону, Мбит/с
1.0a	2003	2,5	250
2.0	2007	5	500
3.0	2010	8	985
4.0	2017	16	1969
5.0	2019	32	3938
6.0 (предложено)	2021	64	7877

Представленная здесь эффективная скорость передачи данных указана для односторонней передачи данных. PCIe поддерживает передачу данных на полной скорости одновременно в обоих направлениях.

Стандарты PCIe поддерживают соединения по нескольким линиям, число которых обозначается как x1, x2, x4, x8, x16 и x32. Большинство современных материнских плат оснащаются как минимум разъемами PCIe x1 и x16. Разъемы PCI x1 совместимы с краевым разъемом платы длиной 25 мм, а для разъемов x16 требуется краевой разъем длиной 89 мм. Карта PCIe будет корректно работать в любом разъеме, в который она физически помещается. Например, карту PCIe x1 можно вставить в разъем x16, и она будет использовать только одну из 16 доступных линий.

Основная область применения разъемов PCIe x16 — интерфейс между процессором и графической картой с целью обеспечения максимальной производительности для приложений с интенсивным использованием графики, таких как игры. Интерфейс PCIe 5.0 x16 обеспечивает одностороннюю передачу данных со скоростью 63 Гбит/с.

В современных компьютерных архитектурах чип процессора обычно обеспечивает прямое соединение с графической картой, установленной в разъем PCIe x16, по 16 линиям PCIe. Это означает, что передавать сигналы графической карты PCIe через чипсет не требуется.

Помимо интерфейсов графической карты и DDR SDRAM, управление большей частью операций ввода-вывода в современных компьютерных системах осуществляет чипсет. Процессор и чипсет взаимодействуют через набор линий HSIO. Чипсет обеспечивает интерфейсы для периферийных устройств, таких как дисковые накопители, сетевые интерфейсы, клавиатура и мышь. Интерфейсы для этих устройств обычно используют стандарты последовательного интерфейса SATA, M.2 и USB, которые обсуждаются далее.

SATA

Serial AT Attachment (SATA) — это стандарт двустороннего последовательного интерфейса для подключения материнских плат компьютеров к устройствам хранения данных. "AT" в аббревиатуре SATA относится к IBM PC AT. Как и одна линия PCIe, интерфейс SATA содержит две пары проводников с дифференциальной передачей сигналов, где по одной паре передаются данные в каждом направлении. В отличие от PCIe, интерфейс SATA предназначен для работы по кабелям, а не по металлическим сигнальным дорожкам на материнских платах. В дополнение к требованиям по электрической части и формату данных стандарт SATA содержит подробные спецификации совместимых кабелей и разъемов.

Кабель SATA содержит одну двустороннюю линию, обеспечивающую связь между процессором и устройством хранения, таким как накопитель на магнитных дисках, дисковод оптических дисков или твердотельный накопитель. В табл. 4.2 приведены основные версии стандарта SATA с указанием года появления и рабочих параметров каждой из них.

Таблица 4.2. *Поколения стандарта SATA*

Поколение SATA	Год появления	Скорость передачи, ГТ/с	Эффективная скорость передачи данных в одну сторону, Мбит/с
1.0	2003	1,5	150
2.0	2004	3	300
3.0	2009	6	600

Скорость передачи данных в этой таблице приведена для односторонней передачи, хотя, как и PCIe, SATA поддерживает полнодуплексную передачу данных.

Совершенствование стандарта SATA продолжается, но по состоянию на 2021 г. объявлений о следующем поколении SATA с более высокой скоростью передачи данных не было.

M.2

В современных **твердотельных накопителях** (solid-state drives, SSD) для хранения данных используется флеш-память, а не вращающиеся жесткие магнитные диски, как в традиционных дисковых накопителях. Из-за радикальных отличий технологии твердотельных накопителей интерфейс SATA, который в большинстве случаев работает достаточно хорошо с накопителями на вращающихся дисках, оказался значительным препятствием для эффективной работы твердотельных накопителей.

Для того чтобы получить доступ к произвольному блоку данных на магнитном диске (называемому **сектором**), головка накопителя должна физически переместиться на дорожку, содержащую этот сектор, затем дождаться, пока начало сектора достигнет положения головки, после чего накопитель сможет начать чтение данных. Напротив, твердотельный накопитель может напрямую обращаться к любому сектору данных, используя способ, очень похожий на доступ процессора к ячейке динамической памяти DRAM.

Спецификация M.2 была разработана для создания компактного и высокопроизводительного интерфейса доступа к флеш-памяти в небольших портативных устройствах. Благодаря этому ограничения интерфейса SATA были устранены, и скорость передачи данных увеличилась в несколько раз по сравнению со скоростью интерфейса SATA.

Помимо работы с запоминающими устройствами M.2 поддерживает и другие интерфейсы, включая PCIe, USB, Bluetooth и Wi-Fi. Современные материнские платы содержат разъемы M.2, которые, помимо более высокой пропускной способности, занимают гораздо меньше места в корпусе компьютера, чем традиционные отсеки для дисковых накопителей.

USB

Интерфейс **USB (Universal Serial Bus)** предоставляет собой простой (с точки зрения пользователя) интерфейс для подключения к компьютерной системе самых разных периферийных устройств. В дополнение к протоколам связи версии стандарта USB определяют требования к кабелям, разъемам и к подаче питания на подсоединенные по кабелю USB устройства.

Кабели USB имеют легко определяемые типы разъемов, а устройства с поддержкой USB поддерживают "горячее" подключение (подключение устройств друг к другу при включенном питании). USB-устройства поддерживают автоматическую настройку при подключении, и в большинстве случаев пользователям не нужно беспокоиться об установке драйверов при подключении нового устройства к компьютеру с помощью USB-кабеля.

Кабели USB ранних версий (до поколения 2.0) содержали одну пару проводов с дифференциальной передачей сигналов, которая единовременно могла передавать данные только в одном направлении. Более поздние версии стандарта USB (начиная с USB 3.2 поколение 1) поддерживают одновременную двустороннюю переда-

чу данных. Кроме того, USB версии 3.2 и USB4 содержат до двух линий, что позволило удвоить скорость передачи данных.

В табл. 4.3 приведены основные версии стандарта USB с указанием даты появления, максимального количества поддерживаемых линий и максимальной скорости передачи данных для каждой из них.

***Таблица 4.3.** Поколения стандарта USB*

Поколение USB	Год появления	Количество линий	Скорость передачи, ГТ/с	Эффективная скорость передачи данных в одну сторону, Мбит/с
1.1	1998	1	0,012	1,5
2.0	2000	1	0,48	60
3.2, поколение 1	2008	1	5	500
3.2, поколение 2	2013	1	10	1200
3.2, поколение 2x2	2017	2	20	2400
USB4	2019	2	40	4800

В поколениях USB до 2.0 обмен данными полностью находится под управлением управляющего компьютера (хоста). Хост инициирует каждую транзакцию обмена данными, отправляя пакеты, адресованные конкретному устройству, и осуществляет передачу данных в устройство или из него. Начиная с USB 3.2 (поколение 1), устройства могут инициировать связь с хостом, что дает подключенным периферийным устройствам возможность генерировать прерывания.

Thunderbolt

Thunderbolt — это набор стандартов высокоскоростных последовательных интерфейсов, представленный в 2011 г. Оригинальный интерфейс Thunderbolt сочетал передачу сигналов PCIe и DisplayPort с использованием двух последовательных линий Thunderbolt.

Thunderbolt 4 — это последнее поколение стандарта Thunderbolt, добавляющее совместимость с USB4 и поддерживающее подключение устройств PCIe и нескольких мониторов высокого разрешения через один компьютерный порт. Thunderbolt 4 использует тот же разъем, что и USB 3.2 и более поздние поколения (разъем USB-C), и поддерживает скорость передачи данных USB4 — 40 Гбит/с. USB-устройства при подключении к порту Thunderbolt 4 работают корректно. Однако периферийные устройства Thunderbolt 4 несовместимы с портами USB-C, отличными от Thunderbolt 4.

В следующем разделе представлен обзор наиболее популярных стандартов интерфейсов графических дисплеев.

Графические дисплеи

В играх, редакторах видеоматериалов, графическом дизайне и анимации производительность обработки видеоданных имеет решающее значение. Создание и отображение графики с высоким разрешением требует огромного объема математических вычислений.

Процессоры общего назначения могут выполнять необходимые для этого вычисления, однако им не хватает производительности, которую ожидают пользователи этих приложений.

Высокопроизводительные графические карты, называемые **графическими процессорами** (graphics processing units, GPU), по сути являются миниатюрными суперкомпьютерами, в значительной степени оптимизированными для выполнения графических вычислительных задач, таких как визуализация трехмерных сцен. Поскольку вычисления, связанные с визуализацией сцен, часто повторяются, с помощью аппаратного распараллеливания можно добиться существенного повышения производительности. Графические процессоры содержат большое количество относительно простых вычислительных блоков, каждый из которых выполняет небольшую часть общей задачи.

Графический процессор может содержать тысячи отдельных процессоров, каждый из которых выполняет функции АЛУ. Первоначальной движущей силой, которая привела к разработке высокопроизводительных графических процессоров, было построение трехмерных сцен, однако более поздние поколения этой технологии нашли широкое применение в таких областях, как анализ больших данных и машинное обучение. Архитектура графического процессора позволяет ускорить решение любой вычислительной задачи с интенсивными вычислениями, которую можно разбить на набор параллельных операций.

Конечно, не всем пользователям нужна сверхвысокая производительность при обработке видеоданных. Для удовлетворения потребностей пользователей со скромными требованиями к работе с графикой и ограниченным бюджетом в современные процессоры часто встраивают графический процессор средней производительности. Во многих приложениях такой подход обеспечивает более чем достаточную производительность обработки графических данных. Такая конфигурация называется **конфигурацией с интегрированной графикой** — это означает, что функция графического процессора интегрирована в кристалл процессора и использует общую системную память совместно с процессором. Компьютерные системы с интегрированной графикой дешевле, но обеспечивают достаточную графическую производительность для решения основных вычислительных задач, таких как работа с электронной почтой, просмотр веб-страниц и видеозаписей.

Многие настольные компьютерные системы, а также некоторые ноутбуки имеют встроенный графический процессор, предлагая возможность установки высокопроизводительной графической карты. Это дает пользователям возможность адаптировать компьютерную систему к своим потребностям по цене и производительности.

В настоящее время для подключения мониторов к компьютерам используется несколько различных стандартов видеосигналов. Поскольку выходные сигналы, генерируемые графическим интерфейсом компьютера, должны быть совместимы с подключенным монитором, компьютеры обычно оснащают видеоразъемами нескольких типов. Компьютерные мониторы и телевизоры высокой четкости обычно также предлагают выбор интерфейсов подключения.

В *главе 6* архитектура средств обработки графических процессоров будет рассмотрена более подробно. В следующих разделах описываются стандарты компьютерных видеоинтерфейсов прошлого и настоящего.

VGA

Видеостандарт **Video Graphics Array (VGA)** для персональных компьютеров был представлен компанией IBM в 1987 г. VGA — это аналоговый интерфейс, который широко применяется и сегодня, хотя большинство современных компьютеров не имеют разъема VGA. Нередко старые компьютеры с видеовыходами VGA используют кабель-переходник для передачи изображения на монитор, поддерживающий видеовход DVI или HDMI.

Современные версии стандарта VGA поддерживают разрешение до 1920 пикселов в ширину и 1200 пикселов в высоту с частотой обновления 60 Гц. Поскольку видеосигнал VGA является аналоговым, при передаче на монитор происходит некоторая потеря качества сигнала. Этот эффект наиболее заметен при высоких разрешениях экрана.

DVI

Видеостандарт **Digital Visual Interface (DVI)** был разработан для улучшения визуального качества компьютерного изображения за счет передачи с компьютера на монитор цифрового видеосигнала. Для обеспечения обратной совместимости со старыми компьютерами и мониторами по кабелям DVI также можно передавать аналоговые сигналы VGA.

Подобно высокоскоростным последовательным интерфейсам, рассмотренным ранее в этой главе, для передачи видеоданных DVI использует последовательную дифференциальную передачу сигналов. Разъем DVI содержит четыре последовательные линии. Отдельные линии несут информацию о красном, зеленом и синем цветах, а по четвертой линии передается общий тактовый сигнал.

Определены три варианта DVI в зависимости от комбинации поддерживаемых типов цифровых и аналоговых видеосигналов.

- **DVI-A** поддерживает только аналоговый видеосигнал. Этот вариант интерфейса предназначен для обеспечения обратной совместимости с компьютерами и мониторами VGA. Разъем DVI-A имеет другое расположение контактов по сравнению с традиционными разъемами VGA, поэтому для подсоединения к устаревшим устройствам VGA требуется кабель-переходник.

- **DVI-D** представляет собой исключительно цифровой интерфейс, поддерживающий одноканальный и двухканальный варианты. Двухканальный вариант предоставляет дополнительные линии последовательной передачи данных, чтобы увеличить пропускную способность для дисплеев с более высоким разрешением. "Двухканальный" не означает, что кабель поддерживает два монитора.
- **DVI-I** представляет собой интегрированный интерфейс, поддерживающий как аналоговый интерфейс DVI-A, так и цифровые режимы DVI-D. Цифровой интерфейс DVI-I может быть как одноканальным, так и *двух*канальным.

Интерфейсы DVI используются в основном в компьютерных мониторах. Эффективная скорость передачи данных одноканального соединения DVI-D составляет 3,96 Гбит/с. Двухканальный DVI-D передает видеоданные вдвое быстрее, чем одноканальный: 7,92 Гбит/с.

HDMI

Мультимедийный интерфейс высокой четкости **High-Definition Media Interface (HDMI)** поддерживается большинством современных компьютеров и мониторов, а также практически всеми современными телевизорами и связанными с ними развлекательными видеоустройствами, такими как DVD-плееры. HDMI поддерживает только цифровые видеоданные (поддержка аналоговых сигналов отсутствует) и использует ту же дифференциальную последовательную шину, что и DVI-D. В дополнение к видеоданным кабели HDMI также передают многоканальный цифровой аудиосигнал.

С момента введения в 2002 г. стандарт HDMI претерпел несколько редакций. Каждая последующая версия поддерживала обратную совместимость, добавляя новые возможности. В поздних версиях стандарта увеличена пропускная способность, расширен диапазон поддерживаемых разрешений экрана, добавлена поддержка звука высокой четкости и Ethernet-подключения по кабелю HDMI, а также функции для поддержки игр. Каждая версия HDMI совместима с прежними версиями, однако новые функции доступны только в конфигурациях, в которых устройство-источник сигнала, устройство отображения и соединительный кабель совместимы с новым стандартом.

Стандарт HDMI версии 2.1 был опубликован в 2017 г. Он поддерживает эффективную скорость передачи данных 42,6 Гбит/с по четырем дифференциальным последовательным линиям.

DisplayPort

Стандарт DisplayPort, введенный в 2006 г., описывает цифровой интерфейс, поддерживающий цифровое видео, аудио и USB-соединения. В то время как стандарт HDMI ориентирован на бытовую электронику, включая телевизоры и домашние кинотеатры, DisplayPort нацелен на вычислительную технику. DisplayPort передает

данные в пакетах со встроенной в каждый пакет информацией о синхронизации, что устраняет необходимость в отдельном канале синхронизации.

Один выход DisplayPort компьютера может управлять несколькими мониторами, соединенными в гирляндную цепь, где один кабель соединяет компьютер с первым монитором, второй кабель соединяет первый и второй мониторы и т. д. Используемые мониторы должны поддерживать такую функцию. Максимальное количество мониторов, которые можно соединить таким образом, ограничено только возможностями видеокарты, максимальной пропускной способностью кабеля, а также разрешением и частотой обновления мониторов.

Версия DisplayPort 2.0, выпущенная в 2019 г., поддерживает эффективную скорость передачи данных до 77,4 Гбит/с по четырем дифференциальным последовательным линиям.

Сетевой интерфейс

Компьютерная сеть — это совокупность цифровых устройств, взаимодействующих через общую среду связи. **Локальная сеть** (local area network, LAN) состоит из ограниченного числа компьютеров, которые могут находиться в одном физическом месте, например в жилом доме или в офисном здании. Подключенные друг к другу компьютеры, телефоны и другие цифровые устройства в вашем доме представляют собой локальную сеть. Для соединения устройств в среде локальной сети можно использовать проводной интерфейс, обычно Ethernet, или беспроводной интерфейс, обычно Wi-Fi.

Территориально удаленные друг от друга компьютеры и локальные сети обмениваются данными с помощью **глобальной сети** (wide area network, WAN). Сервисы глобальной сети часто предоставляются операторами связи, такими как поставщики кабельного телевидения или телефонные компании. Ваша домашняя локальная сеть, вероятнее всего, подключена к Интернету через сервис глобальной сети, предоставляемый вашим оператором телефонной связи или кабельной сети.

Устройства сетевого интерфейса для домашних и корпоративных сетей (**маршрутизаторы**), предоставляемые поставщиками услуг глобальной сети, обычно предлагают поддержку Ethernet или Wi-Fi для подключения локальных устройств к глобальной сети. В следующих разделах представлены технологии Ethernet и Wi-Fi.

Ethernet

Ethernet представляет собой набор сетевых стандартов для соединения компьютеров в локальную сеть с помощью кабелей. Первоначальная версия Ethernet была разработана в 1974 г. Робертом Меткалфом (Robert Metcalfe), работавшим в исследовательском центре Xerox в Пало-Альто. Стандарт Ethernet был запущен в коммерческую эксплуатацию в 1980 г. как технология связи со скоростью 10 Мбит/с для групп компьютеров, соединенных коаксиальным кабелем. Название технологии произошло от исторического термина *luminiferous aether* (светоносный эфир), обо-

значающего гипотетическую среду, заполняющую все пространство и обеспечивающую распространение электромагнитных волн. Кабель Ethernet служит концептуально похожей средой связи.

Институт инженеров по электротехнике и электронике (Institute of Electrical and Electronic Engineers, IEEE) начал разработку стандартов для технологий локальных сетей, включая Ethernet, в 1980 г. Стандарт Ethernet IEEE 802.3 был опубликован в 1985 г. С тех пор он претерпел ряд изменений, включая увеличение скорости передачи данных и добавление различных топологий сети. Наиболее очевидным отличием современных компьютерных сетей от исходного стандарта Ethernet является использование двухточечных кабелей с витыми парами вместо общего коаксиального кабеля.

Современные компьютеры обычно используют интерфейсы Gigabit Ethernet для обмена данными по кабелям с **неэкранированными витыми парами** (unshielded twisted-pair, UTP). Интерфейс Gigabit Ethernet официально определен в стандарте IEEE 802.3ab и поддерживает скорость передачи данных 1,0 Гбит/с (эффективная скорость достигает 99% исходной скорости), хотя количество добавляемых служебных данных значительно варьирует в зависимости от используемого протокола связи.

Передача данных через интерфейс Ethernet осуществляется блоками данных переменного размера, называемых **кадрами**, которые могут содержат до 1518 байт. Заголовок каждого кадра содержит адресную информацию, идентифицирующую интерфейсы Ethernet источника и получателя. Поскольку современные соединения по витой паре являются двухточечными, наиболее распространенной схемой для соединения группы компьютеров является прокладка кабеля от каждого компьютера к коммутатору. **Коммутатор** — это устройство, которое принимает кадры, передаваемые подключенными компьютерами, и на основе адреса получателя, указанного в каждом кадре, сразу же пересылает их правильному получателю. Максимальная рекомендованная длина кабелей Ethernet составляет 100 метров, что ограничивает физический размер локальной сети Ethernet такой областью, как отдельное офисное здание или жилой дом.

Современные материнские платы обычно содержат встроенный интерфейс Ethernet, что избавляет от необходимости использовать разъем PCIe с картой Ethernet. Интерфейс Ethernet, встроенный в материнскую плату или установленный в разъем расширения PCIe, использует одну линию HSIO, соединяющую процессор через чипсет с интерфейсом Ethernet.

Wi-Fi

Институт IEEE выпустил первую версию стандарта беспроводной связи 802.11 в 1997 г. Эта версия поддерживала скорость передачи данных 2 Мбит/с в частотном диапазоне 2,4 ГГц. Стандарт 802.11b, выпущенный в 1999 г. и поддерживавший скорость передачи данных 11 Мбит/с, оказался коммерчески успешным. Эта техно-

логия получила название **Wi-Fi** в 1999 г. с отсылкой к термину "hi-fi", обозначающему высокоточное воспроизведение звука.

Версия 802.11g, выпущенная в 2003 г., поддерживает исходную скорость передачи данных 54 Мбит/с. Версия 802.11n, выпущенная в 2009 г., также поддерживает **технологию использования нескольких передающих и нескольких приемных антенн** (multiple-input-multiple-output, MIMO) и работу в диапазоне 5 ГГц. Стандарт 802.11ac, опубликованный в 2013 г., поддерживает скорость передачи данных в диапазоне 5 ГГц более 500 Мбит/с при использовании конфигураций с антеннами MIMO. Стандарт 802.11ax, утвержденный в 2021 г., обеспечивает увеличение пропускной способности на 39% по сравнению со стандартом 802.11ac для одного клиента и предлагает расширенную поддержку при использовании в средах с высокой плотностью радиосигналов. Версия 802.11ax продвигается под наименованием "Wi-Fi 6" для работы в диапазонах 2,4 и 5 ГГц и "Wi-Fi 6E" — для работы в диапазоне 6 ГГц.

Устройства Wi-Fi могут страдать от помех, создаваемых бытовыми приборами, такими как беспроводные телефоны, микроволновые печи и другие сети Wi-Fi, работающие поблизости. На распространение сигнала Wi-Fi влияют такие факторы, как стены и другие препятствия между передатчиком и приемником, **многолучевое распространение сигнала** (ослабляющая интерференция между сигналом, распространяющимся по прямому пути, и его отраженной копией), а также максимальная мощность, которую разрешено излучать передатчикам Wi-Fi. Использование нескольких антенн в конфигурациях 802.11n, 802.11ac и 802.11ax значительно снижает влияние многолучевого распространения сигналов на качество передачи данных.

Современные маршрутизаторы глобальных сетей, предоставляемые операторами связи, обычно содержат комбинацию интерфейсов Ethernet и Wi-Fi. Основным преимуществом Wi-Fi по сравнению с Ethernet для этих приложений является сокращение количества используемых кабелей.

Одним из недостатков Wi-Fi является возможное нарушение безопасности, поскольку радиочастотный сигнал может распространяться далеко за пределы здания, в котором находятся системы связи. Стандарты Wi-Fi обеспечивают необходимую поддержку безопасности связи с помощью таких протоколов, как **Wi-Fi Protected Access 2 (WPA2)**, но системные администраторы и пользователи должны убедиться в том, что соответствующие функции безопасности включены и секретные данные, такие как пароль доступа к сети, который при этом должен быть достаточно сложным, надежно защищены.

Поддержка Wi-Fi реализована в большинстве портативных мобильных устройств, таких как ноутбуки, смартфоны и планшеты, и непосредственно встроена во многие материнские платы.

В следующем разделе представлены компьютерные интерфейсы с наименьшими требованиями к пропускной способности: клавиатура и мышь.

Клавиатура и мышь

По сравнению с высокоскоростными интерфейсами, рассмотренными ранее в этой главе, требования к пропускной способности для клавиатуры и мыши весьма скромны. Эти устройства поддерживают единственные методы ввода, используемые человеком-оператором в большинстве конфигураций компьютеров, поэтому от них требуется работа со скоростью человеческих действий. Даже самая быстрая машинистка может нажимать клавиши со скоростью не более одного-двух десятков нажатий в секунду.

Клавиатура

Механическая компьютерная клавиатура состоит из набора клавиш, каждая из которых приводит в действие электрический переключатель мгновенного действия. Стандартная полноразмерная клавиатура обычно содержит 104 клавиши, включая клавиши со стрелками, клавиши управления (<Home>, <Scroll Lock> и т. д.) и дополнительную клавиатуру для ввода цифр. Для подсоединения к компьютеру современные клавиатуры обычно используют USB-кабель или беспроводную связь.

Поскольку требования к пропускной способности при взаимодействии с человеком настолько малы, некоторые материнские платы компьютеров снабжены относительно медленным портом USB 2.0 для подключения клавиатуры, одновременно предлагая более быстрые интерфейсы USB 3.2 и выше для высокоскоростных периферийных устройств. Это приводит к небольшому снижению стоимости компонентов материнской платы.

Поскольку клавиатура сообщает о нажатии и отпускании каждой клавиши отдельно, компьютер может обрабатывать одновременные нажатия комбинаций клавиш. Например, результатом нажатия клавиши <A> при удержании нажатой клавиши <Shift> является ввод заглавной буквы *A*.

Некоторые компьютеры и мобильные устройства, такие как планшеты и телефоны, оснащены интерфейсом с сенсорным экраном. Когда на этих устройствах требуется ввод текста, система отображает на экране клавиатуру, и пользователь касается букв, чтобы произвести нажатие клавиш.

Механические клавиатуры, как правило, обеспечивают более точный ввод, и их предпочитают пользователи, вводящие значительное количество текста. Поскольку поверхность сенсорного экрана абсолютно плоская, пальцы пользователя не ощущают нажатия клавиш. Это приводит к более частым ошибкам ввода при использовании клавиатуры на сенсорном экране. И конечно же, при наличии клавиатуры, отображаемой на сенсорном экране, нет необходимости использовать механическую клавиатуру, что является существенным преимуществом для мобильных устройств. Кроме того, ввод с помощью сенсорной клавиатуры не подвержен физическим отказам, которые могут возникать в механических деталях клавиатур. Для пользователей в перчатках ввод текста затруднен как на сенсорных экранах, так и на клавиатурах.

Мышь

Компьютерная мышь — это ручное устройство, которое перемещает указатель по экрану компьютера в горизонтальном и вертикальном направлениях. Пользователь инициирует требуемые действия в зависимости от местоположения указателя, нажимая кнопки мыши. Современные мыши часто имеют небольшое колесико, способное вращаться в любом направлении и используемое для выполнения таких задач, как прокрутка документа.

Как и клавиатура, мышь обычно подключается к компьютеру через интерфейс USB по проводному или беспроводному соединению. Мышь предъявляет низкие требования к пропускной способности, для удовлетворения которых достаточно порта USB 2.0.

Для работы мыши требуется ровная горизонтальная поверхность, обычно столешница, по которой пользователь перемещает мышь. Современные мыши чаще всего используют оптические излучатели и датчики для обнаружения их перемещения по поверхности. Многим моделям мышей трудно работать на сильно отражающих поверхностях, таких как стеклянные столешницы.

Трекбол по своей концепции похож на мышь, за исключением того, что вместо перемещения мыши по поверхности шарик удерживается в фиксированном положении, но его можно вращать в любом направлении с помощью движений руки. Катая шарик вперед, назад, влево и вправо, пользователь может перемещать указатель на дисплее.

Трекболу не требуется столько места на поверхности, сколько нужно мыши, и его можно установить стационарно. Возможность жестко закрепить трекбол делает его предпочтительным указательным устройством для компьютерных станций, установленных на транспортных средствах, включая наземный, морской и воздушный транспорт.

Как и в случае с клавиатурой, компьютер определяет нажатие и отпускание каждой кнопки мыши как отдельные события. Пользователи могут использовать эту возможность для выполнения таких операций, как перетаскивание значка по экрану. Для этого нужно выполнить следующие действия:

1. Наведите указатель на значок.
2. Нажмите и удерживайте левую кнопку мыши.
3. Переместите указатель (с прикрепленным к нему значком) в нужное место.
4. Отпустите кнопку мыши.

Вместе клавиатура и мышь обеспечивают все возможности ввода, необходимые большинству пользователей компьютеров для выполнения интерактивных задач.

В следующем разделе собраны описания подсистем из этой главы для оценки характеристик материнской платы современного компьютера.

Технические характеристики современной компьютерной системы

С помощью информации, содержащейся в этой главе, вы сможете разобраться в большинстве технических характеристик материнской платы, процессора и чипсета современного компьютера. В этом разделе приведен пример характеристик современной (на 2021 г.) материнской платы с некоторыми пояснениями по отдельным функциям.

Разработчики материнских плат компьютеров должны принять ряд решений, например о количестве разъемов для карт расширения PCIe, разъемов DIMM, портов USB и SATA, которые должны быть включены в состав конкретной модели материнской платы. Эти решения принимаются на основе целевой демографической группы потребителей, будь то геймеры, бизнес-пользователи или экономные домашние пользователи.

В качестве примера материнской платы возьмем ASUS Prime X570-Pro (табл. 4.4). Это высокопроизводительная плата, предназначенная для игровых приложений и поддерживающая игровые функции, такие как разгон процессора. **Разгон** — это увеличение тактовой частоты процессора и других компонентов системы с целью повышения производительности.

Разгон также увеличивает тепловыделение и может привести к нестабильной работе устройства, если оно работает на чрезмерной частоте.

Таблица 4.4. *Пример технических характеристик материнской платы*

Компонент	Характеристика	Примечания
Процессор	Гнездо AMD AM4, совместимое с процессорами AMD Ryzen™ третьего поколения	Гнездо содержит 1331 контакт. Процессор взаимодействует напрямую с системной памятью DDR4. Имеется интерфейс PCI 4.0 x16, напрямую связывающий процессор с графическим процессором
Чипсет	AMD X570. 16 линий PCIe 4.0. 6 портов SATA, 6 Гбит/с	Интерфейс между процессором и чипсетом: PCIe 4.0 x4
Графическая карта	До 3 разъемов PCIe 4.0 x16	Возможна параллельная работа нескольких графических процессоров через масштабируемый интерфейс обмена данными Nvidia **Scalable Link Interface (SLI)** или с помощью технологии AMD Crossfire
Разъемы расширения	3 разъема PCIe 4.0 x16. 3 разъема PCIe 4.0 x1	Разъемы x16 можно сконфигурировать как один разъем x16, два разъема x8 или два разъема x8 плюс один разъем x4

Таблица 4.4 *(окончание)*

Компонент	Характеристика	Примечания
Системная память	4 двухканальных разъема для модулей DIMM DDR4 3200 МГц с общей емкостью до 128 Гбайт	Можно установить до четырех модулей DDR4 объемом до 32 Гбайт каждый. Возможен разгон до 5100 МГц
Интерфейсы накопителей	2 разъема M.2. 6 разъемов SATA, 6 Гбит/с	Разъемы M.2 используют интерфейс PCI 4.0 для подключения быстродействующих твердотельных накопителей. Разъемы SATA предназначены для поддержки традиционных дисковых накопителей
Ethernet	1 порт Gigabit Ethernet	Высокоскоростное подключение к сети Ethernet
USB	2 разъема USB 2.0. 1 разъем USB 3.2, поколения 1 (5 Гбит/с). 1 разъем USB 3.2, поколения 2 (10 Гбит/с)	Каждый разъем поддерживает несколько портов USB. Порты USB 2.0 предназначены для клавиатуры и мыши. Порты USB 3.2 предназначены для быстрых периферийных устройств, таких как внешние накопители

Этот пример предназначен для того, чтобы дать некоторое представление о характеристиках компьютеров потребительского класса в верхнем ценовом сегменте по состоянию на 2021 г.



Если вы собираетесь приобрести компьютер, можете использовать информацию из этой главы, чтобы стать более информированным потребителем.

Резюме

Эта глава началась с введения в устройство элементарной ячейки памяти компьютера, состоящей из полевого МОП-транзистора и конденсатора. Мы изучили схему, реализующую битовую ячейку динамической памяти DRAM. Мы рассмотрели архитектуру модулей памяти DDR5 и работу многоканальных контроллеров памяти. Были представлены другие типы устройств ввода-вывода и отдельно рассмотрены высокоскоростные дифференциальные последовательные соединения, широко используемые в таких технологиях, как PCIe, SATA, USB, и видеоинтерфейсы.

Были описаны популярные видеостандарты, включая VGA, DVI, HDMI и DisplayPort. Мы также рассмотрели сетевые технологии Ethernet и Wi-Fi. Затем обсудили стандартные интерфейсы компьютерной периферии, включая клавиатуру и

мышь. Глава завершилась описанием примерных характеристик современной материнской платы с выделением некоторых интересных особенностей.

Благодаря информации, представленной в этой главе, у вас должно сложиться четкое представление о компонентах современных компьютеров, начиная с уровня технических характеристик и заканчивая технологиями, используемыми при реализации схем.

В следующей главе мы рассмотрим функции более высокого уровня, которые должны быть реализованы в компьютерных системах, такие как ввод-вывод для накопителей, обмен данными по сети и взаимодействие с пользователями. Мы изучим уровни программного обеспечения, реализующие эти функции, начиная с уровня набора инструкций и регистров процессора. Будет рассмотрен ряд ключевых аспектов операционных систем, в том числе начальная загрузка, многопоточность и многопроцессорность.

Упражнения

1. Составьте схему реализации логического элемента И-НЕ, используя две пары МОП-транзисторов, объединенных в КМОП-структуры. В отличие от схем вентилей на основе *n-p-n*-транзисторов, для этой схемы не требуются резисторы.
2. 16-гигабитная интегральная схема памяти DRAM имеет два входа выбора группы банков, два входа выбора банка и 17 входов выбора адресов строк. Сколько битов в каждой строке банка в этом устройстве?

5

Аппаратно-программный интерфейс

Бо́льшая часть программного обеспечения компьютеров написана не на уровне инструкций процессора, т. е. не на языке ассемблера. Почти все приложения, с которыми мы ежедневно работаем, написаны на том или ином языке программирования высокого уровня на основе уже существующих библиотек возможностей, которые разработчики приложений использовали в процессе разработки программного обеспечения. Применяемые на практике среды программирования, включающие в себя языки высокого уровня и связанные с ними библиотеки, предлагают множество сервисов, в том числе **функции ввода-вывода** дисковых накопителей, обмен данными по сети и взаимодействие с пользователями, которые легко доступны из программного кода.

В данной главе описываются уровни программного обеспечения, реализующие эти функции, начиная с инструкций процессора в драйверах устройств. Здесь будет рассмотрен ряд ключевых аспектов операционных систем, включая начальную загрузку, планирование процессов, многопоточность и многопроцессорность.

Прочитав эту главу, вы получите представление о сервисах, предоставляемых операционными системами и функциями **базовой системы ввода вывода (BIOS)** и **единого расширяемого интерфейса встроенного ПО** (Unified Extensible Firmware Interface, UEFI). Вы узнаете, как потоки вычислений функционируют на уровне процессора и как несколько процессорных ядер координируют свои действия в компьютерной системе. Вы также получите общее представление о процессе безопасной загрузки операционной системы, начиная с выполнения первой инструкции.

Мы рассмотрим следующие темы:

- драйверы устройств;
- базовая система ввода-вывода;
- процесс загрузки операционной системы;
- операционные системы;

- процессы и потоки;
- многопроцессорность.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Драйверы устройств

Драйвер устройства предоставляет программным приложениям стандартизированный интерфейс для взаимодействия с периферийными устройствами, такими как дисковые накопители. Это избавляет разработчика приложения от необходимости понимать и реализовывать все технические детали, требуемые для надлежащей работы устройств каждого типа. Драйверы устройств также обеспечивают необходимую координацию, когда приложения, написанные разными разработчиками, пытаются одновременно получить доступ к одному и тому же устройству. Большинство драйверов устройств позволяют нескольким приложениям безопасно и эффективно взаимодействовать с несколькими экземплярами соответствующих периферийных устройств.

На самом низком уровне код драйвера устройства предоставляет программные инструкции, которые управляют обменом данными с периферийным устройством, включая обработку прерываний, инициируемых запросами на обслуживание, отправляемыми в устройство. Драйвер устройства управляет работой аппаратных ресурсов процессора, периферийного устройства и других компонентов системы, таких как чипсет процессора.

В компьютерных системах, поддерживающих привилегированные режимы выполнения, драйверы устройств обычно работают с повышенным уровнем привилегий. Код, работающий с повышенным уровнем привилегий, которые обычно ограничены операционной системой и драйверами устройств, имеет полный доступ к возможностям процессора и периферийных устройств. Этот уровень гарантирует предоставление доступа к ресурсам ввода-вывода, недоступным для кода с меньшими привилегиями. Только доверенный код может напрямую взаимодействовать с этими интерфейсами. Если бы код непривилегированного приложения мог получить доступ к аппаратному интерфейсу периферийного устройства, то программная ошибка, вызвавшая неправильное поведение устройства, могла бы оказать непосредственное влияние на все приложения, пытающиеся использовать это устройство. Действия, связанные с переключением потока выполнения инструкций и данных между непривилегированным кодом пользователя и привилегированным кодом драйвера, будут обсуждаться в *главе 9*.

Драйверы, которые выполняются с повышенными привилегиями, называются **драйверами режима ядра**. **Ядро** — это центральный элемент операционной системы, служащий интерфейсом между аппаратными средствами компьютера и функциями операционной системы более высокого уровня, такими как планировщик.

Как мы узнали из *главы 3*, для доступа к устройствам ввода-вывода используются два основных механизма: с распределением памяти и с распределением по портам. В современных компьютерах преобладает ввод-вывод с распределением памяти, однако в некоторых архитектурах, таких как *x*86, по-прежнему поддерживается и используется ввод-вывод с распределением по портам. В архитектуре *x*86 многие современные периферийные устройства предоставляют системный интерфейс, сочетающий оба механизма ввода-вывода.

Инструменты программирования для современных операционных систем, таких как Linux и Windows, предоставляют ресурсы для разработки драйверов устройств, способных взаимодействовать с периферийными устройствами, используя методы ввода-вывода как с распределением по портам, так и с распределением памяти. Установка драйвера устройства в этих операционных системах требует повышенных привилегий, но пользователям драйвера такие привилегии не требуются.

Драйверы современных периферийных устройств могут быть довольно сложными и трудными для понимания теми, кто не слишком хорошо знаком с аппаратной частью и встроенным ПО таких устройств, однако некоторые устройства прежних поколений довольно просты. Одним из примеров таких устройств является параллельный порт принтера, который был введен в ранних моделях персональных компьютеров и в течение многих лет оставался стандартным компонентом ПК. Современные компьютеры редко оснащаются этими интерфейсами, однако недорогие карты расширения с параллельными портами остаются легкодоступными, а современные операционные системы обеспечивают поддержку драйверов для этих интерфейсов. Любители электроники часто используют параллельный порт в качестве простого интерфейса для подключения к ПК под управлением Linux или Windows внешних схем, использующих 5-вольтовые цифровые сигналы **транзисторно-транзисторной логики** (ТТЛ).

В следующем разделе будут рассмотрены детали интерфейса параллельного порта на уровне драйвера устройства.

Параллельный порт

Программный интерфейс для параллельного порта принтера на ПК состоит из трех 8-разрядных регистров, изначально привязанных к последовательным номерам портов ввода-вывода, начиная с шестнадцатеричного значения 378. Этот набор портов обеспечивает работу интерфейса для системного принтера № 1, обозначаемого как LPT1 на IBM PC-совместимых компьютерах под управлением MS-DOS и Windows. Современные ПК могут сопоставлять параллельный порт с другим диапазоном портов ввода-вывода во время инициализации шины **Peripheral Component Interconnect (PCI)**, но в остальном по сравнению с ранними ПК работа этого интерфейса не изменилась.

Драйверы устройств для параллельного порта в современных компьютерах выполняют те же функции, используя те же инструкции, что и в ранних ПК. В этом разделе мы предположим, что порт принтера сопоставлен с традиционным диапазоном портов ввода-вывода в 64-разрядной версии Linux.

Для взаимодействия с аппаратной частью параллельного порта процессор *x*86 выполняет инструкции in и out, обеспечивающие чтение из портов ввода-вывода и запись в них.

Если предположить, что драйвер параллельного порта установлен и инициализирован, то пользовательское приложение может вызвать функцию драйвера для чтения состояния линий передачи данных параллельного порта и записи в них.

Следующие инструкции драйвера считывают дискретные уровни напряжения, присутствующие на восьми линиях передачи данных параллельного порта, и сохраняют полученное 8-битное значение в регистре al процессора:

```
mov edx,0x378
in al,dx
```

На языке ассемблера *x*86 инструкции с двумя операндами записываются в виде опкод адресат, источник. В этом примере использованы регистры al, edx и dx процессора. Регистр al содержит младшие 8 бит 32-разрядного регистра eax, а dx — младшие 16 бит 32-разрядного регистра edx. Эта последовательность инструкций загружает непосредственное значение 0x378 в регистр edx, после чего считывает в регистр al 8-битное значение данных порта, номер которого содержится в dx.

Исходный код на языке С, позволяющий сгенерировать указанные выше инструкции на ассемблере:

```
char input_byte;
input_byte = inb(0x378);
```

Функцию inb, предназначенную для выполнения ввода 8-битного значения из порта ввода-вывода, предоставляет операционная система Linux. Этот код будет функционировать должным образом только в том случае, если он выполняется в этой операционной системе с повышенными привилегиями. Приложение, запущенное на уровне привилегий пользователя, при попытке выполнить эти инструкции потерпит неудачу, поскольку оно не имеет полномочий на выполнение операций ввода-вывода для порта напрямую.

Инструкции для записи байта в регистр данных параллельного порта и тем самым установки состояния восьми дискретных выходных сигналов показаны в следующем фрагменте кода:

```
mov edx,0x378
movzx eax,BYTE PTR [rsp+0x7]
out dx,al
```

Этот код переносит в регистр edx номер порта, затем загружает в регистр eax переменную из стека. rsp — это 64-битный указатель стека. Значение rsp является 64-битным, т. к. драйвер работает в 64-разрядной версии Linux. Опкод movzx означает "перемещение с дополнением нулями", т. е. эта инструкция перемещает 8-битное значение данных (с обозначением BYTE PTR), хранящееся по адресу rsp+0x7, в младшие 8 бит 32-битного регистра eax, заполняя 24 оставшихся бита в регистре eax нулями. Последняя инструкция записывает байт из регистра al по номеру порта, хранящемуся в регистре dx.

Исходный код на языке С, генерирующий эти инструкции:

```
char output_byte = 0xA5;
outb(output_byte,0x378);
```

Аналогично inb, функция outb, предоставляемая ОС Linux, позволяет драйверам устройств записывать 8-битные значения в заданный порт ввода-вывода.

Этот пример демонстрирует механизм взаимодействия между программой, выполняемой процессором, и аппаратными регистрами периферийных устройств на самом низком уровне работы драйвера устройства. Драйверы для более сложных устройств в системах *x*86 обычно сочетают метод ввода-вывода с распределением по портам, показанный в предыдущих примерах, с методом ввода-вывода с распределением памяти, который обращается к интерфейсу устройства с помощью операций чтения и записи, с точки зрения инструкций процессора идентичных обращениям к памяти.

В этих примерах представлены методы доступа к аппаратным средствам, используемые драйверами в оригинальной архитектуре параллельной шины PCI. В следующем разделе рассматриваются функции, благодаря которым традиционные драйверы PCI могут корректно работать на современных компьютерах с PCIe без изменений, в полной мере используя преимущества технологии высокоскоростной последовательной передачи данных PCIe.

Драйверы устройств PCIe

Как было показано в предыдущей главе, PCIe использует высокоскоростные последовательные соединения в качестве каналов связи между процессором и периферийными устройствами с поддержкой PCIe. Возможно, вам интересно, какие действия должен выполнять драйвер устройства для взаимодействия с такими мощными аппаратными средствами. Простой ответ состоит в том, что драйверам не нужно делать ничего особенного, чтобы в полной мере воспользоваться широкими возможностями шины PCIe. Технология PCIe была разработана с учетом требований программной совместимости с параллельной шиной PCI, использовавшейся в ПК 1990-х годов. Драйверы устройств, написанные для PCI, по-прежнему корректно работают на компьютерах, использующих последовательную шину PCIe. Задача перевода инструкций ввода-вывода процессора, таких как in и out, в операции последовательной передачи данных, которую необходимо решить для обмена данны-

ми с устройствами PCIe, прозрачно выполняется подсистемами PCIe в процессоре, чипсете и устройствах с поддержкой PCIe.

Устройства PCI и PCIe автоматически выполняют операцию настройки во время запуска системы и при "горячем" подключении устройства к работающей системе. **"Горячее" подключение** — это установка аппаратного устройства в систему, которая включена.

После завершения настройки интерфейс устройства становится доступен операционной системе. Интерфейс между периферийным устройством PCI или PCIe и процессором может включать в себя любую комбинацию следующих механизмов обмена данными:

- один или несколько диапазонов портов ввода-вывода;
- одна или несколько областей адресов памяти, поддерживающих ввод-вывод с распределением памяти;
- подключение к обработчику прерываний процессора.

Процедура настройки интерфейса применяется как к драйверам PCI, так и к PCIe, позволяя устаревшим драйверам PCI корректно работать в системах с PCIe. Конечно же, интерфейсы физических карт устройства с параллельной шиной PCI и устройства с последовательной шиной PCIe сильно различаются, поэтому сами карты не являются взаимозаменяемыми, т. к. используют шины на основе разных технологий. Разъемы шин PCIe намеренно выполнены отличающимися от разъемов PCI, чтобы предотвратить случайную установку устройств PCI в разъемы PCIe, и наоборот.

Массовая передача данных в периферийные устройства и из них обычно основана на технологии **прямого доступа к памяти** (direct memory access, DMA) — как для PCI, так и для PCIe. В системах с поддержкой PCIe операции DMA в полной мере используют высокие скорости передачи данных, достижимые с помощью многолинейных последовательных соединений, передавая данные через интерфейс со скоростью, близкой к теоретической максимальной скорости, которую может поддерживать каждый одно- или многолинейный канал последовательной связи. Технологическое развитие, вызвавшее вытеснение устаревшей технологии параллельной шины PCI значительно быстродействующей технологией многолинейной последовательной передачи данных PCIe при сохранении полной совместимости с драйверами устройств, было весьма примечательным явлением.

Структура драйверов устройств

Драйвер устройства — это программный модуль, который реализует набор предопределенных функций, позволяющих операционной системе связывать драйвер с совместимыми периферийными устройствами и осуществлять контролируемый доступ к этим устройствам. Это дает возможность системным процессам и пользовательским приложениям выполнять операции ввода-вывода на совместно используемых устройствах.

В данном разделе представлен краткий обзор функций, которые должны быть реализованы в драйвере устройства Linux для использования разработчиками прило-

жений. В этом примере к именам функций в качестве префикса добавляется фиктивное имя устройства mydevice. Пример написан на языке программирования С.

Следующие функции выполняют инициализацию и завершение работы драйвера:

```
int mydevice_init(void);
void mydevice_exit(void);
```

Операционная система вызывает mydevice_init для настройки драйвера устройства при запуске системы или позже, если устройство добавляется в систему посредством "горячего" подключения. Функция mydevice_init возвращает целочисленный код, указывающий, была ли инициализация успешной, или, в случае неудачи, код возникшей ошибки. На успешную инициализацию драйвера указывает возврат нуля.

Когда драйвер больше не нужен, например при завершении работы системы, или когда устройство отсоединяют во время работы системы, система вызывает функцию mydevice_exit, чтобы полностью прекратить доступ к устройству и освободить все системные ресурсы, выделенные для драйвера.

Следующие две функции позволяют системным процессам и пользовательским приложениям инициировать и завершать сеансы связи с устройством:

```
int mydevice_open(struct inode *inode, struct file *filp);
int mydevice_release(struct inode *inode, struct file *filp);
```

Функция mydevice_open пытается инициировать доступ к устройству и сообщает о любых ошибках, которые могут возникнуть при этом. Параметр inode — это указатель на структуру данных, содержащую информацию, необходимую для доступа к определенному файлу или другому устройству. Параметр filp — это указатель на структуру данных, содержащую информацию об открытом файле. В подсистеме ввода-вывода Linux все типы устройств последовательно представляются в виде файлов, даже если само устройство по своей сути не основано на файлах. Имя параметра filp — это сокращение от *file pointer* (указатель файла). Все функции, работающие с данным файлом, получают указатель на эту структуру в качестве входных данных. Среди прочих деталей структура filp указывает, открыт ли файл для чтения, записи или для того и другого.

Функция mydevice_release закрывает устройство или файл и освобождает все ресурсы, выделенные при вызове функции mydevice_open.

После успешного обращения к функции mydevice_open код приложения может начать считывание данных из устройства или запись в него. Функции, выполняющие эти операции:

```
ssize_t mydevice_read(struct file *filp, char *buf,
    size_t count, loff_t *f_pos);
ssize_t mydevice_write(struct file *filp, const char *buf,
    size_t count, loff_t *f_pos);
```

Функция mydevice_read считывает данные из устройства или файла и передает их в буфер в пространстве памяти приложения. Параметр count указывает запрашиваемый объем данных, а параметр f_pos задает смещение от начала файла, с которого следует начать чтение данных. Параметр buf — это адрес места назначения данных. Количество фактически считанных байтов (которое может быть меньше запрошенного числа) предоставляется функцией в виде возвращаемого значения с типом данных ssize_t.

Функция mydevice_write использует большинство тех же параметров, что и mydevice_read, за исключением того, что параметр buf объявляется константой (const), т. к. функция mydevice_write считывает данные, начиная с адреса памяти, указанного параметром buf, и записывает данные в файл или устройство.

Один из интересных моментов в реализации этих функций заключается в том, что привилегированный код драйвера не может (или, по крайней мере, не должен, если операционная система разрешает это) напрямую обращаться к пользовательской памяти. Это сделано для предотвращения случайного или преднамеренного выполнения кодом драйвера считывания или записи в недопустимую область памяти, такую как пространство ядра.

Для того чтобы избежать этой потенциальной проблемы, операционная система предоставляет специальные функции — copy_to_user и copy_from_user, которые следует использовать в драйверах для доступа к пользовательской памяти. Перед копированием данных эти функции выполняют необходимые действия для проверки адресов пользовательского пространства, указанных в вызове функции.

В этом разделе дано краткое введение в операции аппаратного интерфейса, выполняемые драйверами устройств, и представлена общая структура драйвера устройства.

При включении компьютера перед загрузкой операционной системы и инициализацией ее драйверов для проведения низкоуровневого самотестирования и настройки системы необходимо запустить встроенное ПО. В следующем разделе представлено введение в код, который выполняется при включении компьютера прежде других программ: базовая система ввода-вывода.

Базовая система ввода-вывода (BIOS)

Базовая система ввода-вывода компьютера (Basic Input/Output System, BIOS) содержит код, который выполняется при запуске системы. На заре появления персональных компьютеров BIOS предоставляла собой набор программных интерфейсов, которые помогали абстрагироваться от деталей интерфейсов периферийных устройств, таких как клавиатуры и дисплеи.

В современных ПК во время запуска BIOS выполняет тестирование системы и настройку периферийных устройств. После завершения этих действий процессор (под управлением программ) взаимодействует с периферийными устройствами напрямую, без дальнейшего вмешательства BIOS.

В ранних ПК код BIOS хранился в чипе **постоянной памяти** (постоянном запоминающем устройстве, ПЗУ — read-only memory, ROM) на материнской плате. Этот код был жестко запрограммирован и не мог быть изменен. Современные материнские платы обычно хранят BIOS материнской платы в перепрограммируемом устройстве флеш-памяти. Это позволяет устанавливать обновления BIOS для добавления новых функций или устранения проблем, обнаруженных в более ранних версиях встроенного ПО. Процесс обновления содержимого BIOS обычно называют *перезаписью BIOS*.

Один из недостатков перепрограммирования BIOS заключается в том, что эта возможность позволяет внедрять в систему вредоносный код путем записи его во флеш-память BIOS. При успешной реализации атаки этого типа вредоносный код выполняется при каждом запуске компьютера. К счастью, успешные атаки на встроенное ПО BIOS оказались довольно редкими.

Поскольку BIOS берет на себя управление во время запуска системы, одним из первых ее действий является запуск **самотестирования при включении питания** (power-on self-test, POST) для ключевых компонентов системы. Во время выполнения POST-процедуры BIOS пытается взаимодействовать с системными компонентами, включая клавиатуру, дисплей и загрузочное устройство, в качестве которого обычно выступает дисковый накопитель. Компьютер может быть оснащен высокопроизводительным графическим процессором, однако видеоинтерфейс, используемый BIOS во время запуска, может оказаться достаточно примитивным и поддерживать лишь вывод текста.

BIOS использует видеоинтерфейс и интерфейс клавиатуры для отображения любых ошибок, обнаруженных во время тестирования системы, и позволяет пользователю изменять сохраненные настройки. Клавиатура и видеоинтерфейсы, предоставляемые BIOS, позволяют выполнить первоначальную настройку компьютера, в котором еще нет загрузочного устройства.

Если видеоинтерфейс работает неправильно, BIOS не сможет отобразить информацию, связанную с ошибкой. В этой ситуации BIOS пытается использовать динамик ПК, при его наличии, чтобы указать на ошибку, выдавая последовательность звуковых сигналов. Документация по материнской плате содержит информацию о типах ошибок, на которые указывает каждая такая последовательность. Некоторые материнские платы имеют цифровой дисплей для вывода сведений об ошибках в ходе тестов POST и другой информации о состоянии.

В зависимости от конфигурации системы инициализацией устройств PCIe во время запуска системы управляет либо BIOS, либо операционная система. После успешного завершения процесса настройки всем устройствам PCIe назначены совместимые диапазоны портов ввода-вывода, диапазоны ввода-вывода для режима с распределением памяти и номера прерываний.

По мере выполнения процедуры запуска операционная система определяет соответствующий драйвер для связи с каждым периферийным устройством на основе информации о производителе и устройстве, предоставляемой периферийным устройством через интерфейс PCIe. После успешной инициализации драйвер напря-

мую взаимодействует с каждым периферийным устройством для выполнения операций ввода-вывода по запросу. Системные процессы и пользовательские приложения вызывают набор стандартизированных функций драйверов, представленных в предыдущем разделе, для получения доступа к устройству, выполнения операций чтения и записи и закрытия устройства.

Одной из распространенных процедур, связанных с BIOS, которые выполняют пользователи, является выбор порядка загрузки среди доступных устройств хранения данных. Например, эта функция позволяет настроить систему на первую попытку загрузки с оптического диска, содержащего образ операционной системы, если такой диск находится в дисководе. Если загрузочный оптический диск не найден, система может попытаться загрузиться с основного дискового накопителя. Для загрузки операционной системы можно настроить и расположить в приоритетном порядке несколько устройств хранения данных.

Для вызова режима настройки BIOS иногда назначается определенная клавиша, например <Esc> или функциональная клавиша <F2>, которую следует нажать на ранней стадии процесса загрузки. Название соответствующей клавиши обычно отображается на экране вскоре после включения питания. При входе в режим настройки BIOS параметры отображаются в формате меню. Вы можете выбрать один из доступных экранов, чтобы изменить параметры, связанные с такими функциями, как порядок приоритетов загрузки. После внесения изменений в параметры пользователю предоставляется возможность сохранить эти изменения в **энергонезависимой памяти** (nonvolatile memory, NVM) и возобновить процесс загрузки. При этом следует соблюдать осторожность, поскольку внесение ненадлежащих изменений в настройки BIOS может привести к тому, что компьютер не загрузится.

За время, прошедшее с момента появления IBM PC, возможности реализаций BIOS значительно расширились. По мере развития архитектуры ПК и добавления поддержки 32-разрядных, а затем и 64-разрядных операционных систем устаревшая архитектура BIOS не поспевала за потребностями новых, более удобных систем. Крупные участники отрасли выступили с инициативой определить архитектуру встроенного программного обеспечения системы, которая помогла бы избавиться от ограничений BIOS. Результатом этих усилий стал стандарт UEFI, который заменил традиционные возможности BIOS в современных материнских платах.

Единый расширяемый интерфейс встроенного ПО (UEFI)

Единый расширяемый интерфейс встроенного ПО (Unified Extensible Firmware Interface, UEFI) — это выпущенный в 2007 г. стандарт, определяющий архитектуру встроенного ПО, которое реализует функции, предоставляемые устаревшим BIOS, и добавляет ряд существенных улучшений. Как и в случае с BIOS, UEFI содержит код, выполняемый сразу после запуска системы.

Введение UEFI преследует несколько целей, включая добавление поддержки загрузочных дисковых накопителей объемом более 2 **терабайт (Тбайт)**, сокращение времени запуска и повышение безопасности процесса загрузки. UEFI предоставля-

ет несколько функций, которые при включении и правильном использовании существенно снижают вероятность случайного или злонамеренного повреждения встроенного ПО, хранящегося во флеш-памяти UEFI.

В дополнение к возможностям устаревших реализаций BIOS, описанных ранее, UEFI поддерживает следующие функции.

- **Приложения UEFI** — это модули исполняемого кода, хранящиеся во флеш-памяти UEFI. Приложения UEFI расширяют спектр возможностей, доступных в предзагрузочной среде материнской платы, и, в некоторых случаях, предоставляют сервисы для использования операционными системами во время выполнения. Одним из примеров приложений UEFI является оболочка UEFI, которая представляет собой интерфейс командной строки для взаимодействия с процессором и периферийными устройствами. Оболочка UEFI поддерживает запросы данных от устройств и позволяет изменять параметры конфигурации, хранящиеся в энергонезависимой памяти.
- Другим примером приложения UEFI является GNU **GRand Unified Bootloader** (**GRUB**). GRUB поддерживает *многозагрузочные* конфигурации, представляя меню, из которого пользователь может выбрать один из нескольких доступных образов операционной системы для загрузки в ходе запуска системы.
- **Архитектурно-независимые драйверы устройств** предоставляют UEFI доступ к независимым от процессора реализациям драйверов устройств. Благодаря этому можно использовать единую реализацию встроенного ПО UEFI на таких разноплановых архитектурах, как *x*86 и процессоры **Advanced RISC Machine** (**ARM**). Архитектурно независимые драйверы UEFI хранятся в формате байтового кода, который интерпретируется встроенным ПО, соответствующим типу процессора. В ходе загрузки эти драйверы обеспечивают взаимодействие UEFI с периферийными устройствами, такими как графические карты и сетевые интерфейсы.
- **Безопасная загрузка** — использование криптографических сертификатов, гарантирующих, что во время запуска системы выполняются только одобренные драйверы устройств и загрузчики операционной системы. Эта функция проверяет цифровую подпись каждого компонента встроенного ПО, прежде чем разрешить его выполнение. Такая проверка защищает от разнообразных вредоносных атак, нацеленных на встроенное ПО.
- **Ускорение загрузки** достигается за счет параллельного выполнения операций, которые в BIOS выполнялись последовательно. На практике загрузка происходит настолько быстро, что многие реализации UEFI не предлагают пользователю возможность нажимать клавишу во время загрузки, поскольку ожидание ответа ведет к задержке запуска системы. Вместо этого операционные системы, такие как Windows, обеспечивают доступ к настройкам UEFI, предлагая пользователю возможность запрашивать этот доступ во время работы операционной системы с последующей перезагрузкой для отображения экрана настройки UEFI.

UEFI не просто заменяет функции старого BIOS. Это миниатюрная операционная система, поддерживающая расширенные возможности, например позволяющая удаленному специалисту использовать сетевое подключение для устранения неполадок на компьютере, который не загружается.

После тестов POST, низкоуровневой настройки системных устройств и определения соответствующего загрузочного устройства на основе порядка выполнения загрузки система начинает процесс загрузки операционной системы.

Процесс загрузки операционной системы

Процедура загрузки образа системы зависит от структуры (формата) разделов на устройстве хранения данных, где записан этот образ, и функций безопасности, применяемых во время загрузки. Цель процесса загрузки — запустить систему после включения питания и инициализировать операционную систему, оставив компьютер в известном состоянии и готовым к выполнению полезной работы.

С начала 1980-х годов стандартный формат разделов диска назывался **основной загрузочной записью** (master boot record, MBR). Раздел MBR имеет загрузочный сектор, расположенный в логическом начале отведенного для него пространства. Этот загрузочный сектор содержит информацию, описывающую логические разделы устройства. Каждый раздел содержит файловую систему, организованную в виде древовидной структуры каталогов и файлов внутри них.

Из-за фиксированного формата структур данных MBR запоминающее устройство MBR может содержать максимум четыре логических раздела и иметь объем не более 2 Тбайт, т. е. 2^{32} 512-байтовых секторов для размещения данных. Эти ограничения стали весьма ощутимыми, когда объем доступных на рынке дисковых накопителей пересек границу в 2 Тбайт. Для решения этих проблем и параллельно с разработкой UEFI был разработан новый формат разделов — **таблица разделов GUID** (GUID Partition Table, GPT), где **GUID** означает **глобальный уникальный идентификатор** (Global Unique IDentifier). Этот формат устранил ограничения в отношении объема накопителя и количества разделов, предоставив при этом некоторые дополнительные усовершенствования.

Накопитель, размеченный по формату GPT, имеет максимальный размер 2^{64} 512-байтовых секторов, вмещающих более 8 млрд терабайт данных. При обычной настройке GPT поддерживает до 128 разделов на накопитель. Тип каждого раздела указывается 128-битным идентификатором GUID, что позволяет в будущем определять фактически неограниченное количество новых типов разделов. Большинству пользователей не требуется большое количество разделов на одном накопителе, поэтому наиболее очевидным преимуществом GPT для пользователей является поддержка накопителей большего объема.

Процесс загрузки для материнских плат с BIOS и с UEFI имеет некоторые отличия, описанные в следующих разделах.

Загрузка при использовании BIOS

На материнской плате с BIOS после тестов POST и настройки устройств PCIe BIOS инициирует процесс загрузки. BIOS предпринимает попытку выполнить загрузку с первого устройства в настроенной последовательности приоритетов. Если обнаружено допустимое устройство, встроенное ПО считывает из загрузочного сектора MBR небольшой фрагмент исполняемого кода, называемый *загрузчиком*, и передает ему управление. На этом встроенное ПО BIOS завершает свою работу и больше не используется во время функционирования системы. Загрузчик инициирует процесс загрузки и запуска операционной системы.

Если на материнской плате с BIOS используется *диспетчер загрузки*, код загрузочного сектора MBR запускает этот диспетчер вместо непосредственного выполнения загрузки операционной системы. Диспетчер загрузки (например, GRUB) отображает список, из которого пользователь выбирает требуемый образ операционной системы. Встроенное ПО BIOS не располагает информацией о многозагрузочных конфигурациях, и процесс выбора операционной системы в диспетчере загрузки происходит без участия BIOS.

МНОГОЗАГРУЗОЧНАЯ КОНФИГУРАЦИЯ И ПРИОРИТЕТЫ ЗАГРУЗКИ



Многозагрузочная конфигурация в диспетчере загрузки позволяет пользователю выбрать нужную операционную систему из меню доступных вариантов. Это меню отличается от приоритетного списка вариантов загрузки, поддерживаемого BIOS, который позволяет самой BIOS выбирать первый доступный образ операционной системы.

Загрузка при использовании UEFI

На материнской плате с UEFI после завершения тестов POST и настройки устройств (способом, очень похожим на соответствующие действия BIOS) UEFI инициирует процесс загрузки. В ходе процедуры запуска может отображаться диспетчер загрузки. Диспетчер загрузки UEFI, являющийся частью встроенного ПО UEFI, отображает меню, из которого пользователь может выбрать требуемый образ операционной системы.

Если пользователь не выбирает операционную систему из меню диспетчера загрузки в течение нескольких секунд (или если меню диспетчера загрузки не отображается), UEFI пытается выполнить загрузку с первого устройства в настроенной последовательности приоритетов.

Встроенное ПО UEFI считывает исполняемый код диспетчера загрузки (который существует отдельно от диспетчера загрузки UEFI) и файлы загрузчика из настроенных расположений на системном диске, затем выполняет эти файлы в процессе запуска.

На следующей копии экрана показана часть системных **данных конфигурации загрузки** (boot configuration data, BCD), хранящихся в системе Windows 10. Для отображения этой информации на своем компьютере вам следует ввести команду `bcdedit`, используя интерфейс командной строки с правами администратора.

```
C:\>bcdedit
Windows Boot Manager
--------------------
identifier {bootmgr}
device partition=\Device\HarddiskVolume1 path \EFI\MICROSOFT\BOOT\BOOTMGFW.EFI
...
Windows Boot Loader
-------------------
identifier {current} device partition=C:
path \WINDOWS\system32\winload.efi
...
```

В этом примере **диспетчер загрузки Windows** находится в файле `\EFI\MICROSOFT\BOOT\BOOTMGFW.EFI`. Этот файл обычно хранится в скрытом разделе диска и не всегда доступен для отображения в списках каталогов.

Загрузчик Windows идентифицируется как `\WINDOWS\system32\winload.efi` и расположен в каталоге `C:\Windows\System32\winload.efi`.

Вероятность злонамеренной модификации встроенного ПО UEFI или хранящихся на диске программных компонентов, выполняемых во время загрузки системы, требует дополнительной защиты, обеспечиваемой процессом доверенной загрузки.

Доверенная загрузка

Целью процесса доверенной загрузки является выполнение всей последовательности загрузки с гарантией, что все выполняемое программное обеспечение должным образом авторизовано и не было изменено. Загрузочное ПО проверяет подлинность встроенного ПО и файлов программ, вычисляя криптографическую хеш-функцию последовательности байтов, составляющих каждый программный компонент, и подтверждая правильность итогового значения хеш-функции с помощью цифровой подписи.

Криптографическая хеш-функция генерирует "отпечаток пальца" для блока данных произвольной длины. Выходное значение, получаемое с помощью данной криптографической хеш-функции, всегда имеет одну и ту же длину. Например, 256-битный алгоритм безопасного хеширования **Secure Hashing Algorithm (SHA-256)** всегда выдает 256-битный результат независимо от длины входных данных.

Криптографическая хеш-функция обеспечивает безопасность благодаря тому, что любая попытка изменить данные, используемые этой функцией в качестве входных

данных, приведет к тому, что выходные данные хеш-функции также изменятся. Определить набор модификаций блока входных данных, при которых хеш-функция будет выдавать тот же результат, что и при расчете по исходному блоку данных, фактически нереально. В теории можно создать модифицированный набор данных, для которого алгоритм SHA-256 будет выдавать тот же результат, что и исходный блок, однако необходимые для этого вычисления на самом быстром доступном на сегодняшний день суперкомпьютере заняли бы больше времени, чем оставшийся срок существования Земли. Мы обсудим хеш-функции более подробно в *главе 15*.

Для того чтобы создать цифровую подпись, издатель программного обеспечения должен вычислить значение хеш-функции для хранящегося в памяти образа встроенного ПО или программного компонента. Следующий шаг — зашифровать хеш-значение с помощью закрытого ключа. Этот закрытый ключ шифрования связан с общедоступным ключом расшифровки. Зашифрованное хеш-значение формирует цифровую подпись, которая сохраняется вместе с файлом встроенного ПО или программного компонента.

Открытый ключ становится доступным для пользователей встроенного ПО или программного компонента. Этот ключ используется для расшифровки зашифрованного хеш-значения, которое затем можно сравнить с хеш-значением, вычисленным на основе последовательности байтов данных файла во время загрузки. Эти два хеш-значения должны в точности совпадать, что подтвердит факт неизменности файла с момента его подписания цифровой подписью. Как и в случае с криптографическим алгоритмом хеширования, фактически невозможно изменить цифровую подпись таким образом, чтобы скрыть изменения в защищенном файле данных.

Открытый ключ не требуется хранить в тайне, однако пользователи этого ключа должны убедиться, что сам ключ получен из надежного источника и не был заменен злоумышленником. Для того чтобы гарантировать, что в процессе загрузки используются только доверенные открытые ключи, эти ключи хранятся в аппаратном хранилище, недоступном для неавторизованных пользователей и программного обеспечения. Стандартный механизм для этого в ПК называется **доверенным платформенным модулем** (trusted platform module, TPM). По сути, TPM — это микроконтроллер, разработанный специально для защиты криптографических ключей.

На новых компьютерах ключи TPM настраиваются изготовителем системы для обеспечения безопасной загрузки системы с момента ее первоначального включения.

В отличие от персональных компьютеров, большинство мобильных устройств используют гораздо более простой процесс загрузки, который не задействует BIOS или UEFI. В следующем разделе рассматривается процесс загрузки мобильных устройств.

Мобильные устройства

Большинство мобильных устройств, таких как смартфоны, обычно не имеют отдельного встроенного загрузочного ПО, подобного BIOS или UEFI в ПК. Как мы

видели на примере 6502, при подаче питания такие устройства выполняют аппаратный сброс процессора и начинают выполнение кода по указанному адресу. Весь код в этих устройствах обычно находится в энергонезависимой области памяти, такой как флеш-память.

Во время запуска мобильные устройства выполняют последовательность, аналогичную процессу загрузки ПК. Проверка правильности работы и инициализация периферийных устройств проводятся перед их первым использованием. Загрузчику на таких устройствах может потребоваться выбрать один из нескольких разделов памяти, чтобы определить подходящий образ системы. Как и в случае с UEFI, мобильные устройства часто задействуют функции безопасности во время процесса загрузки с целью убедиться, что загрузчик и образ операционной системы являются подлинными, прежде чем разрешить продолжение процесса загрузки.

Как на ПК, так и в мобильных устройствах запуск загрузчика является первым шагом процедуры запуска операционной системы. Далее мы рассмотрим запуск операционной системы.

Операционные системы

Операционная система — это многоуровневый пакет программного обеспечения, обеспечивающий создание среды, в которой приложения выполняют полезные функции, такие как обработка текстов, выполнение телефонных вызовов или управление работой автомобильного двигателя. Приложения, работающие под управлением операционной системы, выполняют алгоритмы, реализованные в виде последовательностей инструкций процессора, и по мере необходимости обмениваются данными с периферийными устройствами с помощью операций ввода-вывода для выполнения своих задач.

Операционная система предоставляет стандартизированные программные интерфейсы, которые разработчики приложений используют для доступа к системным ресурсам, таким как потоки выполнения процессора, файлы на диске, ввод с клавиатуры или других периферийных устройств и вывод на такие устройства, как экран компьютера или приборы на приборной панели автомобиля.

Операционные системы можно разделить на две широкие категории — работающие в реальном времени и работающие не в реальном времени:

- **Операционная система реального времени** (real-time operating system, RTOS) предоставляет функции, гарантирующие, что реакция на входные воздействия последует в течение определенного времени.

  Процессоры, решающие такие задачи, как управление работой автомобильного двигателя или кухонного прибора, обычно работают под управлением операционной системы реального времени. Это служит гарантией того, что электрические и механические компоненты, которыми они управляют, получают отклики на любые изменения входных воздействий в течение строго ограниченного времени.

* **Операционные системы, работающие не в реальном времени**, не гарантируют получения ответа на входные воздействия в заданные сроки. Вместо этого такие системы пытаются выполнить обработку как можно быстрее, даже если иногда для завершения работы требуется много времени.

ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ В СРАВНЕНИИ С ОБЫЧНЫМИ ОПЕРАЦИОННЫМИ СИСТЕМАМИ



Операционные системы реального времени (ОСРВ) не обязательно являются более быстрыми, чем обычные операционные системы, работающие не в реальном времени. Обычная операционная система может быть в среднем быстрее по сравнению с ОСРВ, но при этом такая система может иногда превышать временны́е ограничения, установленные для приложений, выполняющихся в ОСРВ. Цель ОСРВ — никогда не превышать предельное время отклика.

По большей части операционные системы общего назначения, такие как Linux и Windows, не являются операционными системами реального времени. Они стараются выполнить назначенные задачи, вроде загрузки файла в текстовый редактор или вычисления электронной таблицы, как можно быстрее, хотя время выполнения операции может сильно меняться в зависимости от других задач, которые может выполнять система.

Некоторые аспекты операционных систем общего назначения, в частности вывод аудио- и видеосигналов, предъявляют особые требования к работе в реальном времени. Мы все в тот или иной момент наблюдали сбои в воспроизведении видео, при котором изображение на экране прерывается и подвисает. Такой эффект является результатом невыполнения требований дисплея к пропускной способности в реальном времени. Сотовые телефоны предъявляют аналогичные требования к поддержке двустороннего обмена аудиоданными в реальном времени в ходе телефонных вызовов.

Как на стандартных ПК, так и на мобильных устройствах, запуск операционной системы (обычной или работающей в реальном времени), как правило, выполняется с использованием аналогичной последовательности шагов. Загрузчик либо загружает ядро операционной системы в память, либо просто переходит по адресу в виртуальной памяти, чтобы начать выполнение кода операционной системы.

После запуска ядро операционной системы выполняет следующие действия (хотя и не обязательно в указанном порядке).

* Настройка конфигурации процессора и других системных устройств. Сюда относится настройка любых необходимых внутренних регистров процессора и любых связанных с ними устройств управления вводом-выводом, таких как чипсет.

- В системах, использующих виртуальную память со страничной организацией (которая будет представлена в *главе 7*), ядро настраивает модуль управления памятью.
- Запуск системных процессов базового уровня, включая **планировщик** и **процесс простоя**. Планировщик управляет последовательностью выполнения потоков процессов. Процесс простоя содержит код, который выполняется, когда нет других потоков, готовых к запуску планировщиком.
- Составление перечня драйверов устройств и их привязка к каждому периферийному устройству в системе. Для каждого драйвера выполняется код инициализации, как обсуждалось ранее в этой главе.
- Настройка и включение прерываний. После включения прерываний система начинает выполнять операции ввода-вывода с периферийными устройствами.
- Запуск системных сервисов. Эти процессы поддерживают действия, не связанные с операционной системой (например, работу с сетями), а также постоянно действующие установленные функции (например, веб-сервер).
- Для персональных компьютеров запускается процесс пользовательского интерфейса, который отображает экран входа в систему. Этот экран дает пользователю возможность инициировать интерактивный сеанс работы с компьютером. На мобильных устройствах запускается приложение реального времени. Базовая последовательность операций для простого встроенного приложения заключается в считывании входных данных с устройств ввода-вывода, выполнении вычислительного алгоритма для формирования выходных данных и записи этих выходных данных в устройства ввода-вывода с повторением этой процедуры через фиксированные интервалы времени.

В этом разделе термин *"процесс"* используется для обозначения программы, выполняемой процессором. Термин *"поток"* обозначает поток выполнения в пределах процесса, которых может быть несколько. В следующем разделе эти темы рассматриваются более подробно.

Процессы и потоки

Многие (но не все) операционные системы поддерживают концепцию многопоточного выполнения. **Поток** — это последовательность программных инструкций, которая выполняется логически изолированно от других потоков. Операционная система, работающая на одноядерном процессоре, создает иллюзию одновременного выполнения нескольких потоков за счет **разделения по времени**.

При разделении по времени планировщик операционной системы предоставляет каждому готовому к запуску потоку период времени для выполнения. Когда интервал выполнения потока заканчивается, планировщик прерывает запущенный поток и продолжает выполнение следующего потока из своей очереди. Таким образом, планировщик дает каждому потоку немного времени для выполнения, прежде чем вернуться к началу списка и начать цикл заново.

В операционных системах, способных поддерживать несколько выполняемых программ одновременно, термин *"процесс"* относится к запущенному экземпляру компьютерной программы. Для каждого процесса система выделяет ресурсы, такие как память и место в очереди готовых к выполнению потоков планировщика.

При запуске на выполнение процесс состоит из одного потока. По мере выполнения процесс может создавать больше потоков.

Программисты создают многопоточные приложения по разным причинам, включая следующие.

- Один поток может выполнять ввод-вывод, в то время как другой поток выполняет основной алгоритм программы. Например, основной поток может периодически обновлять выводимую на пользовательский дисплей информацию, в то время как другой поток ожидает пользовательского ввода с клавиатуры, находясь в заблокированном состоянии.
- Приложения со значительными требованиями к вычислительным ресурсам могут использовать преимущества многопроцессорных и многоядерных компьютерных архитектур, разделяя большие вычислительные задания на группы более мелких задач, которые можно выполнять параллельно. Выполняя каждую из этих небольших задач в отдельном потоке, программы дают планировщику возможность назначать разные потоки для одновременного выполнения на нескольких ядрах.

В течение своего жизненного цикла процесс проходит через ряд состояний. Вот некоторые состояния процесса, назначаемые операционными системами.

- **Инициализация**: при первом запуске процесса (возможно, двойным щелчком пользователя по значку на рабочем столе) операционная система начинает загружать код программы в память и назначать системные ресурсы для его использования.
- **Ожидание**: после завершения инициализации процесса он готов к запуску. На этом этапе его поток помещается в очередь готовых к выполнению потоков планировщика. Процесс остается в состоянии ожидания до тех пор, пока планировщик не разрешит ему начать выполнение.
- **Выполнение**: поток выполняет программные инструкции, содержащиеся в его разделе кода.
- **Блокировка**: поток переходит в это состояние, когда запрашивает ввод-вывод с устройства, что приводит к приостановке выполнения. Например, блокировку обычно вызывает чтение данных из файла. В заблокированном состоянии поток ожидает завершения обработки запроса драйвером устройства. Как только работающий поток переходит в состояние блокировки, планировщик может переключиться на другой готовый к выполнению поток, пока выполняется операция ввода-вывода первого потока. Когда эта операция завершается, заблокированный поток возвращается в состояние ожидания в очереди планировщика и рано или поздно переходит в состояние выполнения, после чего обрабатывает результаты операции ввода-вывода.

Время выполнения для готовых к запуску процессов распределяет планировщик. Процесс планировщика отвечает за предоставление времени выполнения всем системным и пользовательским потокам.

Планировщик — это управляемая прерываниями процедура, которая выполняется в периодические интервалы времени, а также в ответ на действия потоков, такие как инициирование операций ввода-вывода. Во время инициализации операционной системы к обработчику прерываний планировщика подключается периодический таймер и запускается таймер планировщика.

Пока каждый процесс находится в состоянии инициализации, ядро добавляет в свой список запущенных процессов структуру данных, называемую **блоком управления процессом** (process control block, PCB). Этот блок содержит информацию, необходимую системе для обслуживания процесса и взаимодействия с ним в течение всего его жизненного цикла, включая сведения о выделении памяти и о файле, содержащем исполняемый код процесса. В большинстве операционных систем для идентификации процесса используется целое число, которое остается уникальным в течение всего его жизненного цикла.

В Windows инструмент **Resource Monitor** (Монитор ресурсов) (этот инструмент можно запустить, набрав `Resource Monitor` (Монитор ресурсов) в поле поиска Windows и щелкнув по результату с обозначением **Resource Monitor** (Монитор ресурсов)) отображает информацию о запущенных процессах, включая **идентификатор процесса** (Process IDentifier, PID), назначенный каждому процессу. В Linux команда `top` отображает процессы, потребляющие наибольшее количество системных ресурсов, идентифицируя каждый из них по его идентификатору PID.

Планировщик хранит информацию, связанную с каждым потоком, в **блоке управления потоком** (thread control block, TCB). У каждого процесса есть список связанных TCB, который содержит минимум одну запись. TCB хранит информацию, относящуюся к потоку, такую как **контекст процессора**. Это набор данных, которые ядро использует для возобновления выполнения ожидающего или заблокированного потока. Этот набор состоит из следующих элементов:

- сохраненные регистры процессора;
- указатель стека;
- регистр флагов;
- указатель инструкций.

По аналогии с PID каждый поток имеет **идентификатор потока** (thread identifier, TID), представленный целым числом и являющийся уникальным в течение своего жизненного цикла.

Планировщик использует один или несколько алгоритмов планирования для равномерного распределения времени выполнения между системными и пользовательскими процессами. С первых дней существования вычислительной техники широко используются две основные категории алгоритмов планирования потоков — невытесняющие и вытесняющие:

- **невытесняющие алгоритмы** предоставляют потоку полный контроль над выполнением, позволяя ему выполняться до тех пор, пока он не завершится

или пока он сам не передаст управление планировщику, чтобы другие потоки получили возможность выполнения;

- при использовании **вытесняющих алгоритмов** планировщик имеет право остановить запущенный поток и передать управление выполнения другому потоку без подтверждения со стороны первого потока.

Когда вытесняющий планировщик переключает выполнение с одного потока на другой, он совершает следующие действия:

1. Происходит прерывание по таймеру, которое заставляет планировщик начать выполнение, либо запущенный поток выполняет действие, вызывающее его блокировку, например инициирует операцию ввода-вывода.
2. Планировщик копирует регистры процессора снимаемого потока в контекстные поля блока управления этим потоком (TCB).
3. Планировщик просматривает свой список готовых к выполнению потоков и определяет, какой поток следует перевести в состояние выполнения.
4. Планировщик загружает контекст входящего потока в регистры процессора.
5. Планировщик возобновляет выполнение входящего потока, переходя к инструкции, на которую указывает программный счетчик потока.

Планирование потоков происходит с высокой частотой — это означает, что код, определяющий работу планировщика, должен быть максимально эффективным. В частности, сохранение и извлечение контекста процессора занимает некоторое время, поэтому разработчики операционных систем прилагают все усилия для оптимизации кода переключения контекста планировщика.

В любой момент времени в очереди может быть множество процессов, претендующих на выполнение, поэтому планировщик должен гарантировать, что критические системные процессы будут выполняться с требуемыми интервалами. В то же время, с точки зрения пользователя, приложения должны демонстрировать достаточно быструю реакцию на вводимые пользователем данные, обеспечивая при этом приемлемый уровень производительности при выполнении длительных вычислений.

Для эффективного управления этими противоречивыми требованиями на протяжении многих лет были разработаны различные алгоритмы. Основной особенностью большинства алгоритмов планирования потоков является использование приоритетов процессов. В следующем разделе представлены несколько алгоритмов планирования потоков на основе приоритетов.

Алгоритмы планирования и приоритет процесса

Операционные системы, поддерживающие несколько процессов, обычно предоставляют механизм приоритизации, благодаря которому наиболее важные системные функции получают достаточное время обработки, даже когда система работает с большой нагрузкой, и при этом выделяется достаточное время для выполнения пользовательских процессов с более низким приоритетом. Для различных типов

операционных систем было разработано несколько алгоритмов, преследующих различные цели в области производительности. Вот некоторые алгоритмы планирования, которые были популярны на протяжении многих лет, начиная с самых простых.

- **Первым пришел, первым обслужен** (first come, first served, FCFS). Этот невытесняющий алгоритм был распространен в устаревших операционных системах с пакетной обработкой. В алгоритме планирования FCFS каждому процессу предоставляется контроль над выполнением, и он сохраняет этот контроль до тех пор, пока выполнение не будет завершено. Приоритизация процессов отсутствует, и время завершения любого процесса зависит от времени выполнения процессов, предшествующих ему во входной очереди.
- **Кооперативная многопоточность.** В ранних версиях Windows и macOS использовалась невытесняющая архитектура многопоточности, которая полагалась на допущение, что каждый поток сам передает управление операционной системе через достаточно частые промежутки времени. При этом от разработчиков приложений требовались значительные усилия, чтобы одно приложение не лишало другие приложения возможности выполнения из-за неспособности передать управление через соответствующие промежутки времени. Каждый раз, когда операционная система получала управление, она выбирала следующий поток для выполнения из приоритизированного списка готовых к выполнению потоков.
- **Циклическое планирование.** Вытесняющий циклический планировщик ведет список готовых к выполнению потоков и предоставляет каждому из них интервал выполнения по очереди, возвращаясь в начало списка при достижении его конца. Этот подход фактически устанавливает равные приоритеты для всех процессов, предоставляя каждому из них возможность выполнения в течение определенных периодов времени с интервалами, зависящими от количества процессов в списке планировщика.
- **Планирование с вытеснением и фиксированным приоритетом.** В данном алгоритме каждому потоку присваивается приоритет, указывающий на важность передачи этому потоку управления выполнением, когда он находится в состоянии ожидания. Когда поток, имеющий более высокий приоритет, чем текущий запущенный поток, переходит в состояние ожидания, планировщик немедленно останавливает запущенный поток и передает управление входящему потоку. Планировщик ведет список ожидающих процессов в порядке приоритета, помещая потоки с наивысшим приоритетом в начало очереди. Применение этого алгоритма может привести к тому, что потоки с более низким приоритетом не смогут получить какое-либо время для выполнения, если потоки с более высоким приоритетом монополизируют доступное время выполнения.
- **Частотно-монотонное планирование** (rate-monotonic scheduling, RMS) — это вытесняющий алгоритм с фиксированным приоритетом, обычно используемый в системах реального времени с жесткими сроками завершения (же-

сткий срок завершения — это срок, который нельзя пропустить). При использовании этого алгоритма потокам, которые выполняются чаще, назначаются более высокие приоритеты. Пока выполняются несколько критериев (интервал выполнения потока равен сроку завершения, взаимодействия между потоками, вызывающие задержку, невозможны, а время переключения контекста незначительно), и если максимально возможное время выполнения каждого потока меньше математически рассчитанного предела, сроки завершения гарантированно будут соблюдены.

- **Справедливое планирование** —это планирование, направленное на максимальное использование процессорного времени с гарантией того, что каждому пользователю предоставляется равное количество времени выполнения. Вместо назначения потокам числовых приоритетов фактический приоритет каждого потока определяется количеством времени выполнения, которое он потребил. По мере того как поток использует все больше и больше процессорного времени, его приоритет снижается, предоставляя другим потокам больше возможностей для запуска. Преимущество такого подхода заключается в том, что для интерактивных пользователей, которые не потребляют значительного времени выполнения, быстродействие системы повышается. Ядро Linux использует алгоритм справедливого планирования в качестве планировщика по умолчанию.
- **Многоуровневая очередь с обратной связью.** Этот алгоритм использует несколько очередей, каждая из которых имеет разный уровень приоритета. Новые потоки добавляются в конец очереди с наивысшим приоритетом. В каждом интервале планирования планировщик предоставляет возможность выполнения потоку, стоящему в начале очереди с высоким приоритетом, и удаляет этот поток из очереди, приближая время выполнения остальных потоков. Со временем вновь созданный поток получает возможность выполнения. Если поток потребляет все предоставленное ему время выполнения, он вытесняется по завершении своего интервала и добавляется в конец следующей очереди с более низким приоритетом. Планировщик Windows представляет собой многоуровневую очередь с обратной связью.

Системный процесс простоя содержит поток, который выполняется при отсутствии любого потока, назначенного пользователю или системе и находящегося в состоянии ожидания. Процесс простоя может состоять всего из одной инструкции процессора, задающей бесконечный цикл. Некоторые операционные системы в периоды простоя вместо выполнения цикла простоя переводят систему в режим энергосбережения.

Доля процессорного времени, потребляемого запущенными процессами, вычисляется путем определения доли времени, в течение которого система выполняла активный поток в период измерения. На рис. 5.1 показан Resource Monitor (Монитор ресурсов) Windows, демонстрирующий запущенные процессы, потребляющие наибольшую в среднем долю процессорного времени.

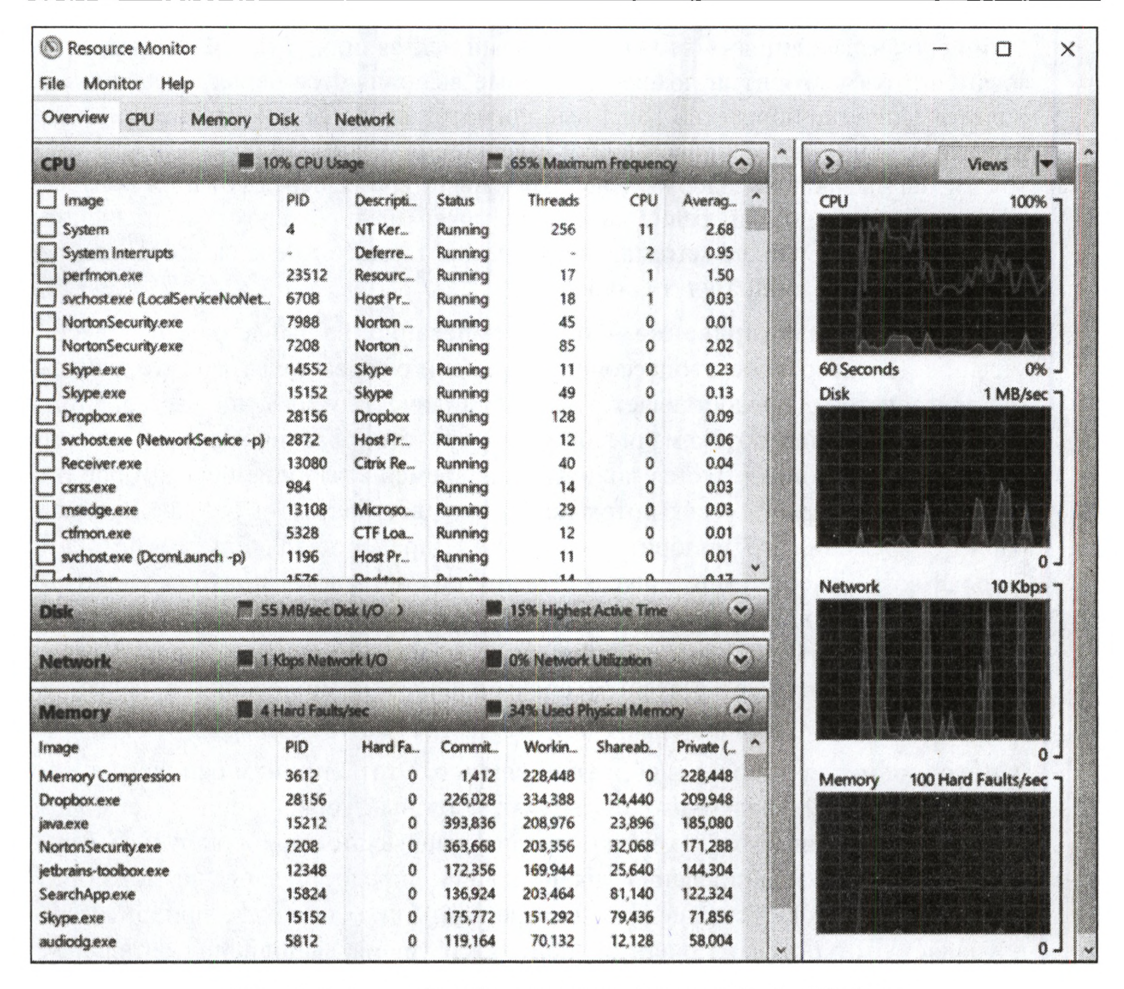


Рис. 5.1. *Экран сведений о процессах в Resource Monitor Windows*

На снимке экрана показано, что в столбце **PID** (ИД процесса) отображаются числовые идентификаторы процессов, а в столбце **Threads** (Потоки) — количество потоков в процессе. Все процессы, показанные на этом экране, находятся в состоянии **Running** (Выполняется).

На рис. 5.2 показан результат выполнения команды `top` в системе Linux.

В верхней части экрана содержится сводная информация, включая количество процессов, называемых здесь задачами (`Tasks`), в каждом из возможных состояний.

Каждая строка в нижней части экрана представляет информацию об одном запущенном процессе. Как и в Windows, в столбце `PID` показаны идентификаторы процессов. Состояние каждого процесса отражено в столбце `S` со следующими возможными значениями:

- `R` — готов к запуску: процесс либо запущен, либо находится в очереди готовых к запуску потоков;

- S — ожидание: приостановлен на время блокировки; ожидание завершения события;
- T — остановлен в ответ на команду управления заданием (это действие выполняется при нажатии комбинации клавиш <Ctrl>+<Z>);
- Z — "зомби": состояние, которое возникает, когда дочерний процесс, относящийся к другому процессу, завершается, но система продолжает выводить информацию об этом дочернем процессе до завершения родительского процесса.

```
jim@jim-VirtualBox: ~
top - 18:16:25 up 11:16,  1 user,  load average: 0.00, 0.21, 0.67
Tasks: 183 total,   1 running, 182 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.8 us,  1.0 sy,  0.0 ni, 94.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4046384 total,  1135716 free,   900736 used,  2009932 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  2798540 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 2110 jim       20   0 1224192 163008  81456 S  3.7  4.0 508:13.96 compiz
  974 root      20   0  358516  79944  36340 S  2.3  2.0  88:58.41 Xorg
 2259 jim       20   0  945448  49416  39064 S  0.7  1.2   0:11.74 nautilus
 9456 jim       20   0  665284  36824  29000 S  0.7  0.9   0:00.84 gnome-terminal-
 1808 jim       20   0  111968   2132   1780 S  0.3  0.1   0:34.57 VBoxClient
 1882 jim       20   0  344984   6452   5376 S  0.3  0.2   0:00.07 ibus-daemon
    1 root      20   0  119876   5992   3980 S  0.0  0.1   0:01.50 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:00.26 ksoftirqd/0
    5 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S  0.0  0.0   0:01.30 kworker/u2:0
    7 root      20   0       0      0      0 S  0.0  0.0   0:01.31 rcu_sched
    8 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S  0.0  0.0   0:00.00 migration/0
   10 root      rt   0       0      0      0 S  0.0  0.0   0:00.10 watchdog/0
   11 root      20   0       0      0      0 S  0.0  0.0   0:00.00 kdevtmpfs
   12 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 netns
   13 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 perf
   14 root      20   0       0      0      0 S  0.0  0.0   0:00.01 khungtaskd
```

Рис. 5.2. Экран с информацией о процессах Linux, выведенной по команде top

В столбце PR отображается приоритет планирования процесса. Меньшие числа обозначают более высокие приоритеты.

До этого момента мы говорили о процессоре компьютера как о единичной сущности. В большинстве современных ПК интегральная схема процессора содержит несколько ядер, каждое из которых реализует функции полноценного независимого процессора, включая блок управления, набор регистров и АЛУ. В следующем разделе обсуждаются атрибуты систем, содержащих несколько процессорных блоков.

Многопроцессорность

Многопроцессорный компьютер содержит два процессора или более, которые одновременно выполняют последовательности инструкций. Процессоры в такой системе обычно совместно используют системные ресурсы, например основную па-

мять и периферийные устройства. Процессоры в многопроцессорной системе могут иметь одинаковую архитектуру или отдельные процессоры могут иметь отличную от других архитектуру для поддержки уникальных системных требований. Системы, в которых все процессоры рассматриваются как равноправные, называют системами с **симметричной многопроцессорной архитектурой**. Устройства, содержащие несколько процессоров в составе одного пакета интегральных схем, называются **многоядерными процессорами**.

На уровне планировщика операционной системы симметричная многопроцессорная среда просто предоставляет больше процессоров для использования в планировании потоков. В таких системах планировщик рассматривает дополнительные процессоры как ресурсы при назначении потоков для выполнения.

В хорошо спроектированной симметричной многопроцессорной системе пропускная способность может приблизиться к идеальному сценарию линейного масштабирования (пропорционально количеству доступных процессорных ядер) при допущении, что конкуренция за общие ресурсы минимальна. Например, если несколько потоков на отдельных ядрах попытаются одновременно получить массовый доступ к основной памяти, неизбежно произойдет снижение производительности, поскольку система распределяет доступ к ресурсам между конкурирующими потоками. Многоканальный интерфейс динамической памяти DRAM может повысить общую производительность системы в этом сценарии.

Симметричная многопроцессорная система является примером архитектуры **с множеством потоков инструкций и множеством потоков данных** (multiple instruction, multiple data, MIMD). MIMD — это конфигурация параллельной обработки, в которой каждое ядро процессора выполняет независимую последовательность инструкций на собственном наборе данных. Конфигурация параллельной обработки **с одним потоком инструкций и множеством потоков данных** (single instruction, multiple data, SIMD), для сравнения, одновременно выполняет одну и ту же инструкцию над несколькими элементами данных.

Современные процессоры реализуют инструкции SIMD для выполнения параллельной обработки больших наборов данных, таких как графические изображения и последовательности аудиоданных. В ПК текущего поколения использование многоядерных процессоров обеспечивает параллелизм выполнения MIMD, а специализированные инструкции внутри процессоров гарантируют определенную степень параллелизма выполнения SIMD. Обработка SIMD будет обсуждаться далее в *главе 8*.

Тактовая частота процессора выросла с 4,77 МГц в оригинальном IBM PC до более чем 4 ГГц в современных процессорах, что почти в тысячу раз больше. В будущем увеличение тактовой частоты, вероятно, будет менее динамичным, поскольку на этом пути перед нами встают фундаментальные физические ограничения. Для того чтобы компенсировать ограниченный прирост производительности за счет увеличения тактовой частоты, процессорная отрасль обратила особое внимание на различные формы параллелизма выполнения в персональных компьютерах и интел-

лектуальных устройствах. В будущем, вероятно, тенденция роста параллелизма продолжится, поскольку системы будут объединять десятки, затем сотни и в конечном счете тысячи процессорных ядер, параллельно работающих на ПК, смартфонах и других цифровых устройствах.

Резюме

Эта глава началась с обзора драйверов устройств, включая подробные сведения о последовательностях инструкций, используемых кодом драйвера для чтения и записи данных в простое устройство ввода-вывода: параллельный порт ПК. Затем мы обсудили системы — устаревшую BIOS и новую UEFI, которые предоставляют код, выполняемый сразу после включения ПК и обеспечивающий тестирование и инициализацию устройства, а также загрузку операционной системы. Мы увидели, как процесс доверенной загрузки помогает гарантировать, что во время запуска системы разрешается выполнение только авторизованного и не подвергшегося изменениям кода.

Затем мы перешли к описанию некоторых фундаментальных элементов операционных систем, включая процессы, потоки и планировщик. Были представлены различные алгоритмы планирования, нашедшие применение в компьютерах прошлого и в современных системах. Мы рассмотрели выходные данные инструментов, реализованных в операционных системах Linux и Windows, которые предоставляют информацию о запущенных процессах.

Глава завершилась обсуждением многопроцессорной обработки и ее влияния на производительность современных компьютерных систем, а также последствий внедрения параллельной обработки в архитектурах MIMD и SIMD для будущего вычислительной техники.

В следующей главе будут представлены различные области, где требуются специализированные вычисления, и их уникальные требования к вычислениям в реальном времени, цифровой обработке сигналов и работе **графического процессора (GPU)**.

Упражнения

1. Перезагрузите компьютер и войдите в раздел настроек BIOS или UEFI. Изучите каждое из доступных в этом разделе меню. Какую систему использует ваш компьютер, BIOS или UEFI? Поддерживает ли ваша материнская плата разгон процессора? Когда вы закончите, обязательно выберите вариант выхода без сохранения изменений, если только вы не обладаете полной уверенностью в том, что хотите внести изменения.
2. Выполните соответствующую команду на компьютере, чтобы отобразить информацию о текущих запущенных процессах. Какой идентификатор **Process ID** (**PID**) имеет процесс, который вы используете для выполнения этой команды?



Специализированные вычисления

Большинство пользователей компьютеров имеют как минимум общее представление о ключевых характеристиках ПК и интеллектуальных цифровых устройств, связанных с производительностью, таких как скорость процессора и объем **оперативной памяти** (random access memory, RAM). Эта глава исследует требования к производительности, свойственные областям вычислений, которые, как правило, менее очевидны для большинства пользователей, включая системы реального времени, цифровую обработку сигналов и обработку данных в **графических процессорах** (graphics processing unit, GPU).

Мы изучим уникальные вычислительные функции, связанные с каждой из этих областей, и рассмотрим примеры современных устройств, реализующих эти функции.

После прочтения этой главы вы будете знать, в каких областях требуются вычисления в реальном времени, и разбираться в основных концепциях цифровой обработки сигналов в контексте ее широкого применения в беспроводной связи. Вы также ознакомитесь с базовой архитектурой современных графических процессоров и с некоторыми современными реализациями компонентов в областях вычислений, обсуждаемых в этой главе.

В этой главе рассматриваются следующие темы:

- вычисления в реальном времени;
- цифровая обработка сигналов;
- обработка данных в графических процессорах;
- примеры специализированных архитектур.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Вычисления в реальном времени

В предыдущей главе было представлено краткое введение в некоторые требования к вычислениям в реальном времени с точки зрения способности системы реагировать на изменения входных данных. Эти требования устанавливаются в виде предельных сроков, ограничивающих время, которое может потребоваться системе для формирования выходных данных в ответ на изменение входных данных. В данном разделе эти временны́е ограничения рассматриваются более подробно, кроме того, здесь представлены некоторые функции, реализуемые в вычислительных системах реального времени для соблюдения временны́х требований.

Вычислительные системы реального времени можно разделить на системы с мягкими и жесткими гарантиями реагирования на изменения входных данных. Считается, что **система мягкого реального времени** работает приемлемо, если она выполняет требования по ожидаемому времени отклика бóльшую часть времени, но не обязательно все время. Примером приложения, работающего в реальном времени, является отображение часов на мобильном телефоне. При открытии окна с изображением часов некоторые реализации этого приложения на мгновение отображают время, которое было показано при последнем просмотре часов, после чего показания сразу же обновляются до правильного текущего времени. Пользователи, безусловно, хотели бы, чтобы часы показывали правильное время каждый раз, когда они отображаются, но кратковременные сбои, подобные этому, обычно не рассматриваются как существенные проблемы.

С другой стороны, **система жесткого реального времени** считается неработоспособной, если она когда-либо пропускает какой-либо из установленных сроков отклика. Критически важные системы безопасности, такие как контроллеры подушек безопасности в автомобилях и системы управления полетом гражданского авиалайнера, имеют жесткие требования к работе в реальном времени. Разработчики таких систем очень серьезно относятся к требованиям по срокам реагирования и прилагают значительные усилия для того, чтобы процессор реального времени удовлетворял этим требованиям при любых возможных условиях эксплуатации.

Поток управления простой системой реального времени показан на рис. 6.1.

На рис. 6.1 представлена вычислительная система реального времени, использующая аппаратный интервальный таймер для управления временнóй последовательностью своей работы.

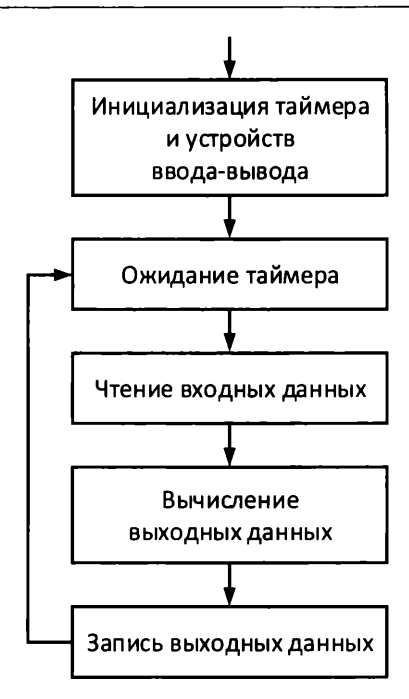


Рис. 6.1. *Поток управления системой реального времени*

Интервальный таймер с обратным отсчетом выполняет повторяющийся цикл следующих действий:

1. Загрузка в регистр счетчика заранее заданного числового значения.
2. Пошаговое уменьшение значения счетчика с фиксированной тактовой частотой.
3. Когда значение счетчика достигнет нуля: формирование события, например установка бита в регистре или вызов прерывания процессора.
4. Возврат к *шагу 1*.

Интервальный таймер генерирует периодическую последовательность событий с точностью, зависящей от характеристик системных часов, для управления которыми часто используют кварцевый кристалл. Ожидая события таймера в начале каждого цикла, система, показанная на рис. 6.1, начинает каждый проход выполнения через фиксированные, почти равные промежутки времени.

Для того чтобы удовлетворить требования жесткого реального времени, время выполнения кода внутри цикла (кода, содержащегося в блоках "Чтение входных данных", "Вычисление выходных данных" и "Запись выходных данных" на рис. 6.1) всегда должно быть меньше, чем интервал таймера. Предусмотрительные разработчики систем внимательно следят за тем, чтобы ни один путь через код не мог довести время выполнения до предела жесткого реального времени. Консервативное правило проектирования системы может состоять в том, что самый длинный путь выполнения кода внутри цикла не должен занимать более 50% интервала таймера.

Практические системы реального времени, построенные в такой конфигурации, могут быть основаны на 8-, 16- или 32-разрядном процессоре, работающем на так-

товой частоте в диапазоне от десятков до сотен мегагерц. Таймер, используемый в основном цикле таких систем, генерирует события с выбранной разработчиком частотой, как правило, в диапазоне от 10 до 1000 Гц.

В менее сложных системах код, представленный блоками на рис. 6.1, часто выполняется непосредственно на аппаратных средствах процессора без задействования промежуточных программных уровней. Эта конфигурация не предусматривает использования операционной системы реального времени, описанной в *главе 5*. С другой стороны, сложное приложение реального времени по всей вероятности будет иметь более широкие потребности, чем те, что можно реализовать с помощью этой упрощенной архитектуры, и это делает привлекательным использование операционной системы реального времени.

Операционные системы реального времени

Операционная система реального времени (ОСРВ) имеет ряд особенностей, внешне схожих с операционными системами общего назначения, рассмотренными в *главе 5*. Однако архитектура ОСРВ значительно отличается от операционных систем общего назначения в том, что все аспекты ОСРВ — от внутренних структур ядра до драйверов устройств и системных сервисов — сосредоточены на удовлетворении требований жесткого реального времени.

В большинстве ОСРВ используется принцип вытесняющей многопоточности, часто называемый в литературе по ОСРВ **многозадачностью**. Термины *"задача"* и *"поток"* в контексте ОСРВ являются в некотором роде синонимами, поэтому для последовательности изложения мы будем продолжать использовать термин *"поток"* для обозначения задач ОСРВ.

Архитектуры ОСРВ на нижнем конце шкалы сложности обычно поддерживают многопоточность в контексте одного прикладного процесса. Эти более простые ОСРВ поддерживают приоритизацию потоков, но часто не имеют функций защиты памяти.

Более сложные архитектуры ОСРВ предоставляют такие функции операционной системы, как защита памяти, в дополнение к вытесняющей многопоточности с приоритизацией. В этих ОСРВ несколько процессов могут одновременно находиться в состоянии *Running* (Выполняется), при этом каждый из процессов может содержать несколько потоков. В системах с защитой памяти доступ к памяти ядра со стороны потоков приложений запрещен, и приложения не могут проникать в области памяти друг друга.

Среды ОСРВ, от низшего до высшего уровня сложности, предоставляют несколько структур данных и методов взаимодействия, направленных на обеспечение эффективного обмена данными между потоками, а также на поддержку контролируемого доступа к общим ресурсам. Эти возможности часто доступны, и не только в ОСРВ. Вот несколько примеров таких возможностей.

- **Взаимное исключение**, или **мьютекс** (*mutex*, от англ. *mutual exclusion*), — это механизм, позволяющий потоку затребовать доступ к общему ресурсу, не

блокируя выполнение других потоков. В своей простейшей форме взаимное исключение реализуется с помощью доступной всем потокам переменной, которая имеет значение 0, когда ресурс свободен, и 1, когда ресурс занят. Поток, которому требуется ресурс, считывает текущее значение такой переменной, и если оно равно 0, присваивает ей значение 1, после чего выполняет операцию с использованием этого ресурса. После завершения операции поток возвращает этой переменной значение 0. В то же время использование взаимных исключений сопровождается рядом следующих потенциальных проблем.

- **Вытеснение потоков.** Допустим, поток считывает переменную взаимного исключения и обнаруживает, что она равна 0. Поскольку планировщик способен прервать выполняющийся поток в любое время, этот поток может быть прерван до того, как он успеет установить для переменной значение 1. Затем возобновляется выполнение другого потока, который берет под контроль тот же ресурс, т. к. видит, что переменная все еще равна 0. Когда возобновляется выполнение первого потока, он заканчивает установку для переменной значения 1 (хотя к этому моменту она уже равна 1). В этот момент оба потока ошибочно полагают, что у них есть исключительный доступ к данному ресурсу, что, скорее всего, приведет к серьезным проблемам, если оба потока попытаются использовать этот ресурс.

  Для того чтобы предотвратить такой сценарий, во многих процессорах реализуют некоторую форму **инструкции "проверить-установить"**. Инструкция "проверить-установить" считывает значение из ячейки памяти и устанавливает для этой ячейки значение 1 за одно непрерываемое действие (также называемое **неделимым**). В архитектуре *x*86 эту операцию выполняет инструкция `BTS` (проверка и установка бита). В архитектурах процессоров, где отсутствует инструкция типа "проверить-установить" (например, 6502), риск вытеснения можно устранить путем отключения прерываний перед проверкой состояния переменной взаимного исключения и последующего включения прерываний после присвоения этой переменной значения 1. Недостатком этого подхода является снижение скорости реагирования в реальном времени в период отключения прерываний.

- **Инверсия приоритетов** происходит, когда поток с более высоким приоритетом пытается получить доступ к ресурсу, в то время как соответствующая переменная взаимного исключения удерживается под контролем потока с более низким приоритетом. В такой ситуации ОСРВ обычно переводят более приоритетный поток в заблокированное состояние, позволяя низкоприоритетному потоку завершить свою работу и освободить переменную.

  Проблема с инверсией приоритетов возникает, когда начинается выполнение потока с приоритетом, который находится между уровнями приоритетов первых двух потоков. Во время работы этого среднеприоритетного потока выполнение низкоприоритетного потока остановлено, и он не может освободить переменную взаимного исключения. Высокоприоритетный по-

ток теперь должен ждать, пока закончится выполнение среднеприоритетного потока, что фактически нарушает всю схему приоритетов потоков. Это может привести к тому, что высокоприоритетный поток не сможет уложиться в заданный срок.

Одним из методов предотвращения инверсии приоритетов является **наследование приоритетов**. В ОСРВ, где реализовано наследование приоритетов, каждый раз, когда поток с более высоким приоритетом (hi_thread) запрашивает доступ к переменной взаимного исключения, удерживаемой потоком с более низким приоритетом (lo_thread), приоритет потока lo_thread временно повышается до уровня hi_thread. Это исключает любую возможность того, что поток со средним приоритетом задержит завершение (первоначально) более низкоприоритетного потока lo_thread. Когда поток lo_thread освобождает переменную взаимного исключения, ОСРВ восстанавливает первоначальный приоритет этого потока.

- **Взаимная блокировка** может возникнуть, когда несколько потоков пытаются заблокировать несколько переменных взаимного исключения. Если потоки thread1 и thread2 требуют контроля над переменными взаимного исключения mutex1 и mutex2, может возникнуть ситуация, в которой поток thread1 блокирует переменную mutex1 и пытается заблокировать переменную mutex2, в то время как поток thread2 уже заблокировал переменную mutex2 и пытается заблокировать переменную mutex1. В такой ситуации ни одна из этих задач не может выполняться, отсюда и термин *"взаимная блокировка"*. Некоторые реализации ОСРВ проверяют принадлежность переменных взаимного исключения при попытке их заблокировать и сообщают об ошибке в ситуации взаимной блокировки. В более простых ОСРВ разработчик системы должен убедиться, что возникновение взаимной блокировки невозможно.

- **Семафор** — это обобщение переменной взаимного исключения. Различают семафоры двух типов: двоичный и подсчитывающий.
  - **Двоичный семафор** похож на взаимное исключение, с той разницей, что он скорее предназначен не для контроля над доступом к ресурсу, а для того, чтобы одна задача могла отправить сигнал другой задаче. Если поток thread1 пытается *взять под контроль* семафор semaphore1, когда он недоступен, thread1 блокируется до тех пор, пока другой поток или процедура обслуживания прерываний не *передаст контроль над* семафором semaphore1.
  - **Подсчитывающий семафор** содержит счетчик с верхним пределом. Подсчитывающие семафоры используются для контроля доступа к нескольким взаимозаменяемым ресурсам. Когда поток берет под контроль подсчитывающий семафор, значение счетчика увеличивается, и задача выполняется. Когда счетчик достигает своего предела, поток, пытающийся взять под контроль семафор, блокируется до тех пор, пока другой поток не отдаст контроль над этим семафором, уменьшив значение счетчика.

Рассмотрим пример системы, которая поддерживает ограниченное количество одновременно открытых файлов.

Для управления операциями открытия и закрытия файлов можно использовать подсчитывающий семафор. Если система поддерживает до 10 открытых файлов и поток пытается открыть 11-й файл, подсчитывающий семафор с пределом 10 будет блокировать операцию открытия файла до тех пор, пока не будет закрыт другой файл и не станет доступен его дескриптор.

- **Очередь** (другое название — **очередь сообщений**) — это однонаправленный путь связи между процессами или потоками. Отправляющий поток помещает элементы данных в очередь, а получающий поток извлекает эти элементы в том же порядке, в котором они были отправлены. ОСРВ синхронизирует доступ между отправителем и получателем, поэтому получатель получает только полные элементы данных. Очереди обычно реализуются с буфером хранения фиксированного размера. Буфер со временем заполняется и блокирует дальнейшее добавление данных, если поток-отправитель добавляет элементы данных быстрее, чем поток-получатель их извлекает.

  Очереди сообщений ОСРВ предоставляют программный интерфейс, позволяющий потоку-получателю проверить, содержит ли очередь данные. Многие реализации очередей также поддерживают использование семафора для подачи сигнала заблокированному потоку-получателю о появлении данных.

- **Критическая секция.** Обычной является ситуация, при которой нескольким потокам требуется доступ к общей структуре данных. При использовании общих данных очень важно, чтобы операции чтения и записи, выполняемые разными потоками, не накладывались во времени друг на друга. Когда такое наложение происходит, считывающий поток может получить противоречивую информацию, если он обращается к структуре данных в то время, когда другой поток выполняет ее обновление. Механизмы взаимного исключения и семафора предоставляют возможности для управления доступом к общим структурам данных. Использование критической секции — это альтернативный подход, который изолирует код, обращающийся к общей структуре данных, и разрешает только одному потоку в любой момент времени выполнять эту последовательность.

  Простым методом реализации критической секции является отключение прерываний непосредственно перед входом в критическую секцию и их последующее включение после выполнения критической секции. Это предотвращает запуск планировщика и предоставляет потоку, обращающемуся к структуре данных, возможность единоличного управления до выхода из критической секции. Однако этот метод имеет недостаток — он снижает скорость реакции в реальном времени, поскольку не позволяет реагировать на прерывания, в том числе выполнять функции планирования потоков, пока прерывания отключены.

Некоторые ОСРВ предлагают более сложную реализацию метода критической секции с использованием объектов данных критической секции. Объекты критической секции обычно предоставляют возможность выбора: поток может либо перейти в заблокированное состояние, пока критическая секция не станет доступной, либо проверить, используется ли критическая секция, без блокировки. Функция проверки доступности критической секции позволяет потоку выполнять другую работу в ожидании, пока критическая секция освободится.

Сегодня в эксплуатации находится гораздо больше вычислительных систем реального времени, чем персональных компьютеров, которые мы воспринимаем как *компьютеры*. Компьютеры общего назначения составляют менее 1% цифровых процессоров, выпускаемых каждый год. Самые разные устройства — от детских игрушек до цифровых термометров, телевизоров, автомобилей и космических кораблей — содержат как минимум один, а зачастую и десятки встроенных процессоров, каждый из которых работает под управлением ОСРВ того или иного типа.

В этом разделе было дано краткое введение в некоторые возможности обмена данными и управления ресурсами, распространенные в различных реализациях ОСРВ. В следующем разделе представлены архитектуры, используемые при обработке цифровых выборок аналоговых сигналов.

Цифровая обработка сигналов

Процессор для цифровой обработки сигналов (digital signal processor, DSP) — это процессор, оптимизированный для выполнения вычислений над оцифрованными представлениями аналоговых сигналов. Используемые в реальном мире сигналы, такие как аудио, видео, **радиочастотные** (РЧ) сигналы сотовых телефонов и сигналы радаров, являются **аналоговыми** по своей природе. Это означает, что обрабатываемая информация — это реакция электрического датчика на непрерывно меняющееся входное напряжение. Прежде чем цифровой процесс может начать работать с аналоговым сигналом, напряжение сигнала должно быть преобразовано в цифровую форму с помощью **аналого-цифрового преобразователя** (АЦП). В следующем разделе описывается работа АЦП и **цифроаналоговых преобразователей** (ЦАП).

АЦП и ЦАП

АЦП измеряет аналоговое входное напряжение и выдает на выходе цифровое слово, представляющее величину входного напряжения. ЦАП выполняет операцию, обратную АЦП, преобразуя цифровое слово в аналоговое напряжение. В процессе преобразования АЦП часто используют внутренний ЦАП.

В применяемых на практике ЦАП используются схемы различных архитектур. Это делается, как правило, с целью получения требуемого сочетания низкой стоимости, высокой скорости и высокой точности. Одной из простейших архитектур ЦАП является **лестница *R*–2*R***, показанная на рис. 6.2 в конфигурации с 4-разрядным входом.

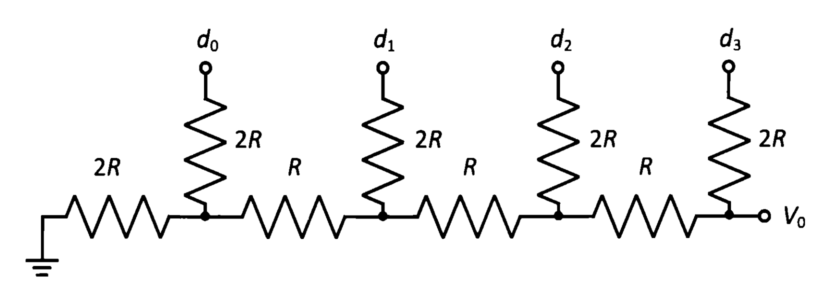


Рис. 6.2. ЦАП лестничного типа R–2R

Этот ЦАП использует 4-разрядное слово данных на входах d_0–d_3 (где d_3 является наиболее значимым битом) для получения аналогового напряжения V_0. Если предположить, что состояние каждого бита 4-разрядного слова d определяется напряжением 0 В (для бита со значением 0) или 5 В (для бита со значением 1), выходное напряжение V равно $(d/2^4) \times 5$ В, где d является значением в диапазоне от 0 до 15. Входное слово со значением 0 дает на выходе напряжение 0 В, а входное слово со значением 15 — $(15/16) \times 5$ В = 4,6875 В. Промежуточные значения слова d дают на выходе равномерно распределенные напряжения с шагом $(1/16) \times 5$ В = 0,3125 В.

АЦП может использовать внутренний ЦАП, подобный описанному выше (хотя обычно с большей разрядностью и, соответственно, более высоким разрешением по напряжению), вместе со схемой выборки-хранения для определения цифрового эквивалента аналогового входного напряжения. Поскольку аналоговый входной сигнал может меняться непрерывно с течением времени, схемы АЦП обычно используют схему **выборки-хранения** для поддержания постоянного аналогового входного напряжения в процессе преобразования. Схема выборки-хранения представляет собой аналоговое устройство с дискретным входным сигналом удержания. Когда вход удержания неактивен, выход схемы повторяет входное напряжение. Когда вход удержания активен, схема выборки-хранения фиксирует и удерживает выходное напряжение на уровне входного напряжения, которое присутствовало в момент появления сигнала удержания.

При постоянном сигнале на выходе схемы выборки-хранения АЦП использует свой ЦАП для определения цифрового эквивалента входного напряжения. Для этого АЦП использует **компаратор**, представляющий собой схему, которая сравнивает два аналоговых напряжения и выдает дискретный выходной сигнал, указывающий, какое из двух напряжений выше. АЦП подает выходное напряжение схемы выборки-хранения на один вход компаратора, а выходное напряжение ЦАП — на другой его вход, как показано на рис. 6.3, где размер входного слова ЦАП равен n битам.

Задача АЦП заключается в нахождении входного слова ЦАП, которое приводит к изменению состояния компаратора. Простой способ сделать это — вести отсчет от нуля, записывая каждое числовое значение на вход ЦАП и наблюдая за выходом компаратора, чтобы понять, изменилось ли его состояние. Выходной сигнал ЦАП, который первым вызовет изменение состояния компаратора, является наименьшим выходным напряжением ЦАП, которое больше выходного напряжения схемы выборки-хранения. Фактическое зафиксированное аналоговое напряжение находится

между этим значением выходного напряжения ЦАП и его значением, когда на вход ЦАП подается слово данных на один отсчет меньше. АЦП такой конфигурации называют **АЦП последовательного счета**.

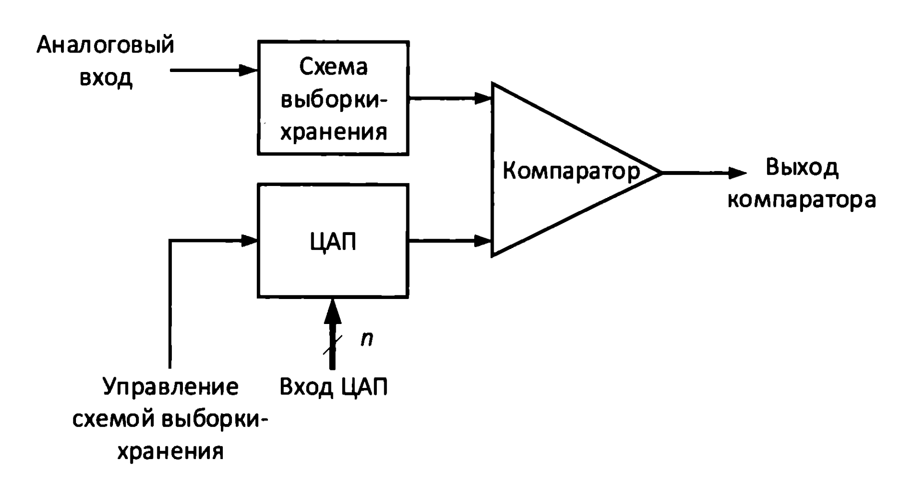


Рис. 6.3. *Архитектура АЦП*

Несмотря на простоту концепции, АЦП последовательного счета может быть довольно медленным, особенно при большом размере слова. Более быстрый метод заключается в последовательном сравнении каждого бита в слове данных ЦАП, начиная с наиболее значимого бита (d_3). Если начать со слова данных 1000b для нашего примера с 4-разрядной схемой, то первое показание компаратора позволит определить местоположение уровня аналогового входного напряжения — выше или ниже средней точки диапазона напряжения ЦАП. Таким образом определяется значение бита d_3 показания АЦП: 0 или 1. Используя известное значение d_3, устанавливаем d_2 равным 1, чтобы определить, в пределах какой четверти полного диапазона находится входное напряжение. Эта процедура повторяется для последовательного определения каждого из оставшихся битов и заканчивается на младшем значащем бите.

Такой метод преобразования в АЦП называется **последовательным приближением**. АЦП последовательного приближения работает намного быстрее, чем АЦП последовательного счета. В нашем примере максимально возможное количество сравнений уменьшается с 16 до 4. В 12-разрядном АЦП последовательного приближения максимально возможное число сравнений уменьшается с 4096 до 12. В общем случае, использование метода последовательного приближения вместо последовательного счета уменьшает максимальное количество шагов для n-разрядного АЦП с 2^n до n. АЦП последовательного приближения имеют разрешение от 8 до 18 бит и максимальную частоту преобразования до нескольких мегагерц.

АЦП и ЦАП характеризуются разрешающей способностью и максимальной скоростью преобразования. Разрешение АЦП или ЦАП определяется количеством битов в его слове данных. Максимальная скорость преобразования определяет, насколько быстро АЦП или ЦАП может выдавать последовательные выходные данные.

При обработке данных в реальном времени АЦП выполняет последовательность измерений через периодические промежутки времени, получая входные данные для дальнейшей обработки. Требования к разрешению слов данных и частоте дискретизации сильно различаются в зависимости от конкретного применения DSP. Вот некоторые примеры стандартных форматов оцифрованных аналоговых данных.

- Цифровой звук для компакт-дисков дискретизируется с частотой 44,1 кГц и 16 битами на выборку в двух каналах, соответствующих левому и правому динамикам.
- Видеокамеры измеряют аналоговый сигнал интенсивности света, полученный от каждого пиксела двумерной матрицы, и преобразуют полученные показания в цифровое слово размером, как правило, 8 бит. Отдельные, расположенные близко друг к другу датчики с цветовыми фильтрами производят измерения для красного, зеленого и синего цветов в каждом пикселе изображения. Полный набор данных для одного пиксела насчитывает 24 бита, состоящих из трех 8-битных значений цвета. Одно изображение может содержать десятки миллионов пикселов, а видеокамеры обычно выдают от 30 до 60 кадров в секунду. Так как цифровое видео производит огромный объем данных, для снижения требований к их хранению и передаче обычно используются алгоритмы сжатия.
- Мобильный телефон содержит радиочастотный приемопередатчик, который преобразует принятый радиосигнал в частотный диапазон, подходящий для ввода в АЦП. Типичные параметры для АЦП мобильного телефона: 12-битное разрешение и частота дискретизации 50 МГц.
- Автомобильная радарная система отбирает радиочастотную энергию, отраженную от близлежащих препятствий, с разрешением 16 бит и частотой 5 МГц.

В следующем разделе мы рассмотрим требования к обработке сигналов для последовательностей данных, захваченных АЦП.

Особенности аппаратных средств DSP

Процессоры цифровой обработки сигналов (DSP) оптимизированы для выполнения алгоритмов обработки цифровых выборок аналоговой информации. **Скалярное произведение** — это фундаментальная операция, используемая во многих алгоритмах, выполняемых DSP. Если $\vec{a}$ и $\vec{b}$ представляют собой два вектора равной длины (**вектор** — это одномерный массив числовых значений), то их скалярное произведение вычисляется путем умножения каждого элемента $\vec{a}$ на соответствующий элемент $\vec{b}$ и суммирования полученных произведений. С математической точки зрения, если длина каждого вектора равна n (с индексами от 0 до $n-1$), то скалярное произведение векторов равно:

$$\vec{a}\cdot\vec{b}=\sum_{i=0}^{n-1}a_ib_i=a_0b_0+a_1b_1+a_2b_2+\ldots+a_{n-1}b_{n-1}.$$

Повторяющийся характер вычисления скалярного произведения указывает естественный путь для оптимизации производительности в цифровых системах. Основная операция, выполняемая при вычислении скалярного произведения, называется **умножением с накоплением** (multiply accumulate, MAC).

Одна операция MAC состоит из перемножения двух чисел и добавления результата в аккумулятор, который был инициализирован путем обнуления перед вычислением скалярного произведения. Математическая производительность чипов DSP обычно измеряется в количестве операций MAC в секунду. Многие архитектуры DSP способны выполнить одну операцию MAC за один тактовый цикл выполнения инструкций.

Для того чтобы выполнять такую операцию в каждом такте, DSP не может выделять отдельные такты для чтения инструкции MAC из памяти программ, чтения каждого из умножаемых векторов из памяти данных, вычисления произведения и добавления его в аккумулятор. Все эти действия необходимо выполнить за один шаг.

Архитектура фон Неймана, представленная в *главе 1*, использует единую область памяти для программных инструкций и данных. Такая конфигурация приводит к ограничению, известному как **узкое место архитектуры фон Неймана**, возникающему в результате необходимости передавать как инструкции программ, так и значения данных через единый интерфейс "процессор — память".

Это ограничение можно устранить с помощью архитектуры, которая разделяет хранилище программ и данных на две отдельные области памяти, каждая из которых имеет собственный интерфейс для обмена данными с процессором. В такой конфигурации, называемой **гарвардской архитектурой**, программные инструкции и операции доступа к памяти данных можно выполнять параллельно, благодаря чему требуется меньшее число тактов. Гарвардская архитектура будет обсуждаться более подробно в *главе 7*.

Процессор DSP с гарвардской архитектурой должен выполнить две операции обращения к памяти данных для извлечения элементов векторов $\vec{a}$ и $\vec{b}$, которые требуется перемножить с помощью операции MAC. Обычно для этого требуются два такта, что не позволяет достичь целевого показателя производительности в одну операцию MAC за один такт. **Модифицированная гарвардская архитектура** поддерживает использование памяти программ для хранения значений данных в дополнение к инструкциям. Во многих приложениях DSP значения одного из векторов (скажем, вектора $\vec{a}$), участвующих в операции скалярного произведения, являются постоянными величинами, известными на момент компиляции приложения. В модифицированной гарвардской архитектуре считываемые из АЦП в качестве входных данных элементы вектора $\vec{a}$ могут храниться в памяти программ, а элементы вектора $\vec{b}$ — в памяти данных.

Для выполнения каждой операции MAC в этой архитектуре один элемент вектора $\vec{a}$ считывается из памяти программ, один элемент вектора $\vec{b}$ считывается из памяти данных, а подлежащее накоплению произведение сохраняется во внутреннем

регистре процессора. Если DSP содержит кеш-память для программных инструкций, инструкция MAC, выполняющая каждый шаг для вычисления скалярного произведения, будет извлечена из кеша, как только первая операция MAC считает ее из памяти программ, что позволит исключить дальнейшие циклы доступа к памяти для ввода этой инструкции. Такая конфигурация (модифицированная гарвардская архитектура с кешированием программных инструкций) позволяет использовать выполняемые за один цикл операции MAC для всех итераций нахождения скалярного произведения после завершения первой операции MAC. Поскольку векторы, участвующие в реальных вычислениях скалярного произведения, обычно содержат сотни или даже тысячи элементов, общая производительность операций вычисления скалярного произведения может достаточно близко подойти к идеалу — одной операции MAC за такт DSP.

Процессоры DSP можно классифицировать по признаку поддержки их архитектурой операций с фиксированной и плавающей запятой. Процессоры DSP с фиксированной запятой для выполнения математических операций, таких как MAC, используют целые числа со знаком или без знака. Процессоры DSP с фиксированной запятой, как правило, дешевле, чем процессоры DSP с плавающей запятой. Однако математика с фиксированной запятой может привести к проблемам с числами, таким как переполнение, которое может проявляться в превышении диапазона аккумулятора результатов скалярных произведений.

Для того чтобы уменьшить вероятность переполнения, для поддержки вычислений скалярного произведения длинных векторов в DSP часто используют аккумулятор расширенного диапазона, например шириной 40 бит в 32-разрядной архитектуре. Из-за опасений по поводу переполнения и связанных с этим проблем программирование DSP с фиксированной запятой требует дополнительных усилий для исключения вероятности того, что эти эффекты приведут к неприемлемому снижению производительности.

В DSP с плавающей запятой для внутренних вычислений часто используют 32-разрядные числа. Если DSP получает целочисленное значение АЦП, вся дальнейшая обработка выполняется с использованием операций с плавающей запятой. За счет использования преимуществ операций с плавающей запятой вероятность возникновения таких проблем, как переполнение, значительно снижается, что позволяет сократить циклы разработки программного обеспечения.

Использование обработки с плавающей запятой также повышает достоверность результатов вычислений, что выражается в улучшении **отношения "сигнал/шум"** (signal-to-noise ratio, SNR) по сравнению с эквивалентной реализацией с фиксированной запятой. Вычисления с фиксированной запятой квантуют результат каждой математической операции на уровне младшего значащего бита целого числа. Операции с плавающей запятой, как правило, дают точные результаты каждой операции, при этом их точность находится в пределах незначительной доли соответствующего младшего значащего бита числа с фиксированной запятой.

Алгоритмы обработки сигналов

Основываясь на нашем понимании аппаратных средств DSP и операций, которые они поддерживают, далее мы рассмотрим некоторые примеры алгоритмов цифровой обработки сигналов в реальных приложениях.

Свертка

Свертка — это формальная математическая операция наравне со сложением и умножением. В отличие от сложения и умножения, которые работают с парами чисел, свертка работает с парами сигнальных векторов. В контексте DSP сигнал представляет собой серию оцифрованных выборок меняющегося во времени входного сигнала, отобранных через равные промежутки времени. Свертка — это самая фундаментальная операция в области DSP.

Во многих практических приложениях одним из двух сигналов, участвующих в операции свертки, является фиксированный вектор чисел, хранящихся в памяти. Другой сигнал представляет собой последовательность выборок, полученных в результате измерений с помощью АЦП.

Для того чтобы реализовать операцию свертки, при получении каждого измерения АЦП процессор DSP вычисляет измененный выходной сигнал, который является скалярным произведением фиксированного вектора данных (допустим, длина этого вектора равна n) и самых последних входных выборок в количестве n, полученных от АЦП. Для того чтобы вычислить свертку этих векторов, каждый раз при получении выборки от АЦП процессор DSP должен выполнить n операций MAC.

Фиксированный вектор в этом примере, обозначаемый $\vec{h}$, называется **импульсной характеристикой**. Цифровой импульс определяется как теоретически бесконечная последовательность отсчетов, в которой один отсчет равен 1, а все предыдущие и последующие отсчеты равны 0. Использование этого вектора в качестве входных данных для свертки с вектором $\vec{h}$ приводит к получению выходных данных, идентичных последовательности $\vec{h}$, окруженной предыдущими и последующими нулями. Единственное значение 1 в импульсной последовательности умножает каждый элемент $\vec{h}$ в последовательных итерациях, в то время как все остальные элементы $\vec{h}$ умножаются на 0.

Конкретные значения, содержащиеся в векторе $\vec{h}$, определяют влияние операции свертки на последовательность входных данных. Цифровая фильтрация является одним из распространенных применений свертки.

Цифровая фильтрация

Частотно-избирательный фильтр представляет собой схему или алгоритм, который принимает входной сигнал и передает его составляющие в нужных частотных диапазонах на выход без искажений, при этом устраняя или, по крайней мере, уменьшая до приемлемого уровня его составляющие в частотных диапазонах за пределами нужных диапазонов.

Мы все знакомы с регуляторами низких и высоких частот в аудиосистемах. Это примеры частотно-избирательных фильтров. Функция баса реализует **фильтр нижних частот** с переменным усилением. Это означает, что аудиосигнал фильтруется для выбора его низкочастотной части, после чего этот отфильтрованный сигнал подается на усилитель, который изменяет свою выходную мощность в зависимости от положения регулятора баса. Секция высоких частот реализована аналогичным образом, с использованием **фильтра верхних частот** для выбора более высоких частот в звуковом сигнале. Выходы этих усилителей объединяются для получения сигнала, посылаемого на динамики.

Частотно-избирательные фильтры можно реализовать с помощью аналоговой технологии или методов цифровой обработки сигналов. Простые аналоговые фильтры дешевы и требуют лишь нескольких схемных компонентов. Однако рабочие характеристики этих простых фильтров оставляют желать лучшего.

Одними из основных параметров частотно-избирательного фильтра являются **затухание в полосе задерживания** и **ширина переходной полосы**. Затухание в полосе задерживания показывает, насколько хорошо фильтр справляется с устранением нежелательных частот на выходе. Как правило, фильтр не устраняет нежелательные частоты полностью, но для практических целей эти частоты можно уменьшить до настолько малого уровня, что они перестают иметь какое-либо значение.

Переходная полоса фильтра — это диапазон частот между полосой пропускания и полосой задерживания. **Полоса пропускания** — это диапазон частот, которые должны быть пропущены через фильтр, а **полоса задерживания** — это диапазон частот, которые должны быть заблокированы фильтром. Невозможно обеспечить идеально четкую границу между полосой пропускания и полосой задерживания. Между этими полосами имеется некоторая переходная полоса, и чтобы сделать ее как можно более узкой, требуется более сложный фильтр, чем фильтр с более широкой переходной полосой.

Цифровой частотно-избирательный фильтр реализуется с помощью операции свертки с использованием тщательно подобранного набора значений для вектора $\vec{h}$. При правильном выборе элементов $\vec{h}$ можно спроектировать фильтры верхних и нижних частот, полосно-пропускающие и полосно-заграждающие фильтры. Как обсуждалось в предыдущих абзацах, фильтры верхних и нижних частот пропускают высокие и низкие частоты, соответственно, блокируя другие частоты. **Полосно-пропускающий фильтр** пропускает только частоты в пределах указанного диапазона и блокирует все частоты вне этого диапазона. **Полосно-заграждающий фильтр** пропускает все частоты, кроме тех, которые находятся в указанном диапазоне.

Цели высокоэффективного частотно-избирательного фильтра — обеспечить минимальное искажение сигнала в полосе пропускания, эффективную блокировку частот в полосе задерживания и минимальную ширину переходной полосы.

Для создания высокоэффективного аналогового фильтра может потребоваться сложная схема, включающая дорогостоящие прецизионные компоненты. С другой стороны, высокоэффективный цифровой фильтр реализуется с помощью всего лишь операции свертки. Цифровая схема, реализующая высокоэффективный фильтр

нижних частот с минимальными искажениями в полосе пропускания и узкой переходной полосой, может потребовать длинный вектор $\vec{h}$, содержащий, возможно, сотни или даже тысячи элементов. Решение о реализации такого фильтра в цифровом виде зависит от доступности недорогих ресурсов DSP, способных выполнять операции MAC со скоростью, требуемой назначением фильтра.

Быстрое преобразование Фурье

Преобразование Фурье, названное в честь французского математика Жана-Батиста Жозефа Фурье, раскладывает сигнал во временно́й области на набор синусоидальных и косинусоидальных волн различных частот и амплитуд. Исходный сигнал можно восстановить путем суммирования этих волн посредством процесса, называемого **обратным преобразованием Фурье**.

Процессоры DSP работают с отсчетами сигналов временно́й области, отбираемыми через фиксированные интервалы. Благодаря такому отбору реализация преобразования Фурье в DSP называется **дискретным преобразованием Фурье** (ДПФ). В общем случае ДПФ преобразует последовательность из *n* равноотстоящих друг от друга временны́х отсчетов функции в последовательность из *n* отсчетов ДПФ, равноотстоящих друг от друга по частоте.

Каждый отсчет ДПФ представляет собой компле́ксное число, состоящее из действительного числа и мнимого числа. **Мнимое число** при возведении в квадрат дает отрицательный результат.

Мы не будем здесь углубляться в математику мнимых чисел. Альтернативный способ рассмотрения комплексного числа, представляющего частотную составляющую ДПФ (называемую **частотным элементом**), заключается в том, чтобы рассматривать действительную часть комплексного числа как множитель для косинусоидальной волны на частоте частотного элемента, а мнимую часть — как множитель для синусоидальной волны на той же частоте. Суммирование этих волновых составляющих дает представление во временно́й области для данного частотного элемента ДПФ.

Простейшая реализация алгоритма ДПФ для последовательности длины *n* — это двойной вложенный цикл, в котором каждый из циклов выполняет итерацию *n* раз. Если требуется увеличить длину ДПФ, то количество математических операций возрастает пропорционально квадрату *n*. Например, для вычисления ДПФ сигнала длиной в 1000 отсчетов требуется не менее миллиона операций.

В 1965 г. Джеймс Кули (James Cooley) из IBM и Джон Тьюки (John Tukey) из Принстонского университета опубликовали работу, описывающую компьютерную реализацию более эффективного алгоритма ДПФ, который стал известен как **быстрое преобразование Фурье** (БПФ). Описанный ими алгоритм был первоначально изобретен немецким математиком Карлом Фридрихом Гауссом около 1805 г.

Алгоритм БПФ разбивает ДПФ на меньшие ДПФ, где длины меньших ДПФ могут быть перемножены для получения количества отсчетов в исходном ДПФ. Повышение эффективности, обеспечиваемое алгоритмом БПФ, является наибольшим, когда

длина последовательности для ДПФ равна степени числа 2, что позволяет выполнять рекурсивное разложение надвое по всей длине ДПФ. БПФ для 1024 точек требует всего лишь нескольких тысяч операций по сравнению с более чем миллионом операций для реализации двойного вложенного цикла ДПФ.

Важно понимать, что БПФ работает с той же последовательностью входных данных, что и ДПФ, и выдает те же выходные данные, что и ДПФ, но БПФ делает это *намного* быстрее для более длинных последовательностей.

БПФ используется для решения многих практических задач при обработке сигналов. Вот некоторые примеры.

- **Спектральный анализ.** Выходные данные ДПФ для последовательности отсчетов временно́й области — это набор компле́ксных чисел, отражающих амплитуду синусоидальных и косинусоидальных волн в диапазоне частот, представляющем сигнал. Эти амплитуды прямо указывают, какие частотные компоненты присутствуют в сигнале со значительными уровнями, а какие частоты вносят меньший или пренебрежимо малый вклад.

  Спектральный анализ используется в таких областях, как обработка аудиосигналов, изображений и радиолокационных сигналов. Лабораторные приборы, называемые **анализаторами спектра**, обычно используются для тестирования и контроля работы радиочастотных систем, таких как радиоприемники и радиопередатчики.

  Анализатор спектра показывает периодически обновляемое изображение, представляющее частотный состав входного сигнала, полученное с помощью БПФ, вычисленного по отсчетам этого сигнала.

- **Банки фильтров** представляют собой наборы отдельных частотно-избирательных фильтров, каждый из которых обрабатывает отдельную полосу частот. Полный набор фильтров в банке охватывает весь диапазон частот входного сигнала. Примером банка фильтров является **графический эквалайзер**, используемый в высококачественной аудиоаппаратуре.

  Банк фильтров на основе БПФ полезен для разложения нескольких разделенных по частоте каналов данных, передаваемых в виде единого комбинированного сигнала. В приемнике БПФ разделяет принятый сигнал на несколько полос, каждая из которых содержит независимый канал передачи данных. Сигнал, содержащийся в каждой из этих полос, дополнительно обрабатывается для извлечения его содержимого.

  Использование банков фильтров на основе БПФ широко распространено в радиоприемниках для систем широкополосной цифровой передачи данных, таких как цифровое телевидение и мобильная связь 5G.

- **Сжатие данных.** сигнал можно сжать до меньшего размера путем выполнения БПФ и отбрасывания частотных составляющих, считающихся несущественными. Остальные частотные компоненты образуют меньший набор данных, который может быть дополнительно сжат с использованием стандартных методов кодирования.

Такой подход называется **сжатием с потерями**, поскольку часть информации входного сигнала теряется. Сжатие с потерями обычно приводит к большей степени сжатия сигнала по сравнению со **сжатием без потерь**. Алгоритмы сжатия без потерь используются в ситуациях, когда любая потеря данных недопустима, например при сжатии компьютерных файлов данных.

- **Дискретное косинусное преобразование** (**ДКП**) аналогично по концепции ДПФ, за исключением того, что вместо разложения входного сигнала на набор синусоидальных и косинусоидальных функций, как в ДПФ, при ДКП входной сигнал раскладывается только на косинусные функции, каждая из которых умножается на действительное число. Вычисление ДКП можно ускорить с помощью того же подхода, что используется в БПФ для ускорения вычисления ДПФ.

  ДКП обладает тем ценным свойством, что во многих задачах сжатия данных бо́льшая часть информации о сигнале представлена в меньшем количестве коэффициентов ДКП по сравнению с альтернативными алгоритмами, такими как ДПФ. Это позволяет отбросить большее количество менее значимых частотных составляющих, повысив тем самым эффективность сжатия данных.

Сжатие данных с помощью ДКП используется во многих прикладных областях, с которыми ежедневно взаимодействуют пользователи компьютеров и аудиовизуальных развлекательных систем, включая аудио в формате MP3, изображения в формате **JPEG** (Joint Photographic Experts Group) и видео в формате **MPEG** (Moving Picture Experts Group).

Процессоры DSP часто используют для решения задач, связанных с обработкой одно- и двумерных данных. Примерами одномерных данных являются аудиосигналы и радиочастотные сигналы, принимаемые приемопередатчиком мобильного телефона. Одномерные данные сигналов состоят из значения одного отсчета в каждый момент времени для каждого из нескольких входных каналов.

Фотографическое изображение служит примером двумерных данных. Двумерное изображение описывается его шириной и высотой в пикселах и количеством битов, представляющих каждый пиксел. Каждый пиксел изображения отделен от окружающих пикселов пространственным смещением по горизонтали и вертикали. Каждый пиксел на изображении (теоретически) отбирается в один и тот же момент времени.

Видеоизображение представляет собой трехмерную информацию. Один из способов определения видеосегмента — это последовательность двумерных изображений, следующих друг за другом через равные промежутки времени. Традиционные процессоры DSP оптимизированы для работы с двумерными данными одного изображения, однако это не означает, что они оптимальны и для обработки последовательных изображений с высокой частотой обновления.

В следующем разделе описываются графические процессоры, т. е. процессоры, специализированные для удовлетворения требований, связанных с синтезом и отображением видео.

Обработка данных в графических процессорах

Графический процессор — это цифровой процессор, оптимизированный для выполнения математических операций, связанных с генерированием и визуализацией графических изображений для их отображения на экране компьютера. Основными областями применения графических процессоров являются воспроизведение видеозаписей и создание синтезированных изображений трехмерных сцен.

Производительность графического процессора измеряется с учетом разрешения экрана (ширина и высота изображения в пикселах) и частоты обновления изображения в кадрах в секунду. Воспроизведение видео и визуализация сцен — это сложные процессы жесткого реального времени, в которых любое нарушение плавного и регулярного обновления изображения через равные промежутки времени, скорее всего, будет воспринято пользователями как неприемлемое подвисание графики.

Как и видеокамеры, которые мы упоминали ранее в этой главе, графические процессоры обычно представляют каждый пиксел с помощью трех 8-битных значений цвета, указывающих интенсивность красной, зеленой и синей составляющих. Любой воспринимаемый цвет можно получить путем комбинирования соответствующих значений каждого из этих трех цветов. В каждом цветовом канале значение 0 указывает на отсутствие цвета, а 255 — на максимальную интенсивность. Черный представлен тремя нулевыми значениями цвета (красный, зеленый, синий) = (0, 0, 0), а белый — (255, 255, 255). 24 бита цветовых данных дают возможность отобразить более 16 млн уникальных цветов. Степень отличия между соседними 24-битными значениями цвета, как правило, меньше, чем может различить человеческий глаз.

В современных персональных компьютерах функции графического процессора доступны в различных конфигурациях:

- графическая карта может быть установлена в разъем PCIe;
- система может содержать графический процессор в виде одной или нескольких отдельных интегральных схем на основной плате процессора;
- функции графического процессора могут быть встроены в интегральную схему центрального процессора.

Самые мощные графические процессоры потребительского класса реализованы в виде карт расширения с интерфейсом PCIe. Эти высокопроизводительные графические процессоры содержат выделенную графическую память и имеют быстрый канал связи с основным системным процессором (обычно с использованием разъема PCIe x16) для приема команд и данных, представляющих отображаемую сцену. Некоторые графические процессоры поддерживают использование нескольких идентичных карт в одной системе для создания сцен на одном графическом дисплее. Эта технология требует применения отдельной высокоскоростной шины, соединяющей графические процессоры друг с другом. Использование нескольких графических процессоров в системе обеспечивает эффективное распараллеливание графической обработки.

Графические процессоры используют концепцию **параллелизма данных** для одновременного выполнения идентичных вычислений над вектором элементов данных, создавая соответствующий вектор выходных данных. Современные графические процессоры поддерживают тысячи одновременно выполняющихся потоков, реализуя возможность визуализации сложных трехмерных изображений, содержащих миллионы пикселов, со скоростью 60 кадров в секунду или более.

Архитектура типичного графического процессора состоит из одного или нескольких многоядерных процессоров, каждый из которых поддерживает многопоточное выполнение алгоритмов параллельной обработки данных. Интерфейс между графическими процессорами и графической памятью оптимизирован таким образом, чтобы обеспечить максимальную среднюю пропускную способность передачи данных, а не свести к минимуму задержку доступа (что является целью проектирования основной системной памяти). В графических процессорах допускается некоторое увеличение задержки ради достижения максимальной скорости потоковой передачи данных между графическим процессором и его выделенной памятью, поскольку максимальная пропускная способность обеспечивает достижение максимально возможной частоты обновления кадров.

В компьютерных системах с менее жесткими требованиями к производительности графической обработки, таких как компьютеры бизнес-класса, более дешевым и часто вполне приемлемым решением является интеграция менее производительного графического процессора в одну микросхему с основным процессором. Интегрированные графические процессоры могут воспроизводить потоковое видео и реализуют в некоторой степени ограниченный уровень возможностей трехмерной визуализации сцен по сравнению с графическими процессорами более высокого класса.

Вместо выделенной графической памяти интегрированные графические процессоры используют для визуализации графики часть системной памяти. Такой подход приводит к снижению производительности, однако подобные системы обеспечивают достаточную производительность обработки графики для решения большинства офисных и домашних задач.

Интеллектуальные устройства, такие как мобильные телефоны и планшеты, также содержат графические процессоры, обеспечивающие те же возможности воспроизведения видео и трехмерной визуализации сцен, что и более крупные персональные компьютерные системы. Требования компактности и пониженного энергопотребления неизбежно ограничивают производительность графических процессоров мобильных устройств. Тем не менее современные смартфоны и планшеты в полной мере способны воспроизводить потоковое видео высокой четкости и отображать сложную игровую графику.

Графические процессоры как процессоры обработки данных

В течение многих лет архитектуры графических процессоров разрабатывались исключительно для поддержки вычислительных потребностей визуализации трех-

мерных сцен в реальном времени. В последние годы пользователи и производители графических процессоров все чаще признают, что эти устройства на самом деле являются миниатюрными суперкомпьютерами, пригодными для решения гораздо более широкого спектра задач. Современные графические процессоры обеспечивают скорость выполнения операций с плавающей запятой, измеряемую триллионами операций с плавающей запятой в секунду (**терафлопс**). По состоянию на 2021 г. типичный высокопроизводительный графический процессор поддерживал производительность операций с плавающей запятой на уровне десятков терафлопсов, и мог выполнять математические алгоритмы параллельной обработки данных в сотни раз быстрее, чем стандартный настольный компьютер.

Используя преимущества огромной мощности параллельных вычислений, доступной в высокопроизводительных графических процессорах, производители этих устройств предоставляют программные интерфейсы и расширенные аппаратные возможности для реализации более обобщенных алгоритмов. Безусловно, графические процессоры, даже с улучшениями для поддержки требований вычислений общего назначения, по-настоящему эффективны только при ускорении алгоритмов, которые предполагают распараллеливание обработки данных.

Далее приведены некоторые области применения, которые оказались подходящими для ускорения операций с помощью графических процессоров.

Большие данные

В таких разноплановых сферах, как моделирование климата, картирование генов, бизнес-аналитика и анализ сейсмических данных, области решаемых проблем имеют общую потребность в максимально эффективном анализе огромных объемов данных, часто измеряемых в **терабайтах (Тбайт)** или **петабайтах (Пбайт)** (1 Пбайт = 1024 Тбайт). Во многих случаях алгоритмы такого анализа перебирают большие наборы данных в поисках тенденций, корреляций и более сложных связей между элементами данных, которые на первый взгляд могут показаться несопоставимыми и не связанными между собой массивами выборок.

До недавнего времени анализ таких наборов данных при соответствующем уровне детализации часто отвергался как неосуществимый из-за слишком большого времени, необходимого для выполнения такой обработки. Однако сегодня многие приложения для обработки больших данных дают результаты за вполне разумные периоды времени, комбинируя использование графических процессоров (часто на машинах, содержащих несколько взаимосвязанных графических процессоров) и распределяя задачу между несколькими компьютерными системами в облачной среде. В наши дни использование нескольких компьютеров, каждый из которых содержит несколько графических процессоров, для выполнения алгоритмов с высокой степенью параллелизма над огромным набором данных может потребовать удивительно низких затрат по сравнению с затратами, которые обычно связывают с суперкомпьютерами.

Глубокое обучение

Глубокое обучение — это категория методов искусственного интеллекта (ИИ), которая использует многоуровневые сети искусственных нейронов для моделирования фундаментальных операций клеток человеческого мозга. Биологический **нейрон** — это тип нервной клетки, которая обрабатывает информацию. Нейроны связаны между собой посредством **синапсов** и используют для обмена информацией электрохимические импульсы. Во время обучения человеческий мозг настраивает связи между нейронами, чтобы кодировать усваиваемую информацию для последующего извлечения. Человеческий мозг содержит десятки миллиардов нейронов.

Искусственные нейронные сети (artificial neural network, ANN)) используют программную модель поведения нейронов для имитации процессов обучения и извлечения информации, протекающих в человеческом мозге. Каждый искусственный нейрон получает входные данные от (возможно) многих других нейронов и вычисляет один числовой результат. Некоторые нейроны управляются непосредственно входными данными, подлежащими обработке, а другие производят выходные данные, получаемые в результате вычислений ANN. Каждый канал связи между нейронами имеет связанный с ним весовой коэффициент, который представляет собой обычное число, умножающее силу сигнала, проходящего по этому пути. Выраженные в виде числа входные данные для нейрона — это сумма получаемых им входных сигналов, каждый из которых умножен на вес соответствующего пути.

Нейрон вычисляет выходной сигнал, используя формулу, называемую **функцией активации**. Она определяет степень влияния входных данных на "срабатывание" каждого нейрона.

На рис. 6.4 представлен пример одного нейрона, который суммирует входные данные от трех других нейронов (N_1–N_3), умножая их на весовые коэффициенты (w_1–w_3). Полученная сумма передается в функцию активации $F(x)$, которая вырабатывает выходной сигнал нейрона.

Использование трех входных сигналов в этом примере является произвольным; в реальных задачах каждый нейрон может получать входные данные от любого количества других нейронов.

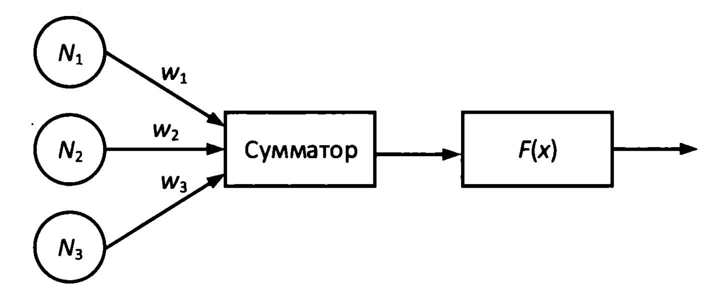


Рис. 6.4. Нейрон, получающий входные сигналы от трех других нейронов

ИНС имеет многоуровневую организацию, в которой за первым уровнем, называемым **входным уровнем**, следует один или несколько внутренних уровней (называемых **скрытыми уровнями**), после которых следует **выходной уровень**. Некоторые ИНС организованы в виде последовательности потоков данных, протекающих от входа к выходу. Сеть такой конфигурации называют **сетью прямого распространения**. Другие конфигурации предусматривают обратную связь от некоторых нейронов к нейронам на предыдущих уровнях. Сеть такой конфигурации называют **рекуррентной сетью**.

На рис. 6.5 показан пример простой сети прямого распространения с тремя входными (input) нейронами, скрытым (hidden) уровнем, состоящим из четырех нейронов, и двумя выходными (output) нейронами. Данная сеть является полносвязной. Это означает, что каждый нейрон на входном и скрытом уровнях связан со всеми нейронами на следующем уровне. Веса соединений на этой диаграмме не показаны.

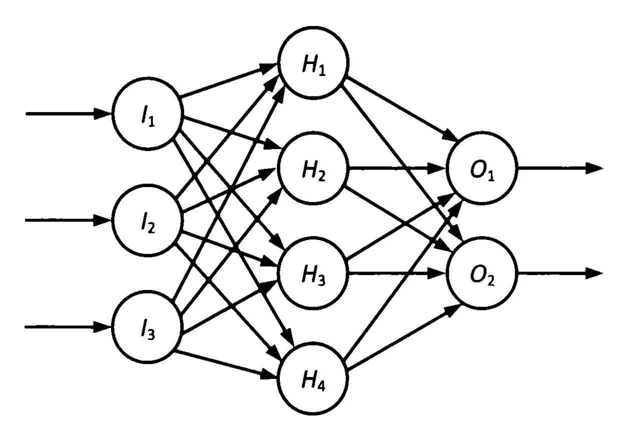


Рис. 6.5. *Трехуровневая сеть прямого распространения*

Обучение ИНС состоит в настройке взвешенных связей между нейронами таким образом, чтобы при представлении определенного набора входных данных получался желаемый результат.

Применяя соответствующий алгоритм обучения, ИНС можно обучить с помощью набора данных, состоящего из известных заведомо правильных выходных данных для широкого спектра входных данных.

Обучение большой и сложной ИНС выполнению сложной задачи, такой как управление автомобилем или игра в шахматы, требует огромного количества обучающих итераций, взятых из очень большого набора данных. Во время обучения каждая итерация вносит небольшие корректировки в весовые коэффициенты внутри сети, постепенно приводя сеть в состояние конвергенции. После полной конвергенции сеть считается обученной и может быть использована для получения выходных данных при представлении новых входных данных. Другими словами, сеть обобщает информацию, которую она усвоила во время обучения, и применяет эти знания к новым ситуациям.

Обработка на графическом процессоре очень хорошо подходит для ИНС благодаря их параллельному характеру. Человеческий мозг фактически представляет собой компьютер с миллиардами независимых процессорных блоков и массовым параллелизмом. Эта форма параллелизма используется на этапе обучения ИНС для ускорения конвергенции сети путем параллельного выполнения вычислений, связанных с множеством искусственных нейронов.

В следующем разделе будут представлены некоторые примеры типов компьютерных систем, реализованных на основе архитектурных концепций, которые были описаны в этой главе.

Примеры специализированных архитектур

В этом разделе рассматриваются некоторые конфигурации вычислительных систем, специализированные для решения определенных задач, и выделяются особые требования к архитектуре, свойственные каждой из них. Мы рассмотрим следующие конфигурации.

- **Сервер для облачных вычислений.** Несколько поставщиков предлагают клиентам доступ к вычислительным платформам через Интернет. Пользователи могут загружать программные приложения на соответствующий облачный сервер и выполнять любые виды вычислений по своему желанию. Как правило, эти сервисы выставляют счета своим клиентам на основании типа и количества выделенных вычислительных ресурсов и продолжительности их использования. Преимуществом для клиента является то, что за эти услуги не надо платить, когда они не используются.

  На более высоком уровне производительности клиенты могут использовать серверы, содержащие несколько соединенных друг с другом карт с графическими процессорами, для выполнения крупномасштабных вычислений с интенсивным использованием операций с плавающей запятой на огромных массивах данных. В контексте облачных вычислений вполне естественным и часто экономически выгодным является подход с разбиением вычислительных задач на более мелкие части, которые могут выполняться параллельно на нескольких серверах с графическими процессорами.

  Благодаря этому организации и даже частные лица с ограниченными финансовыми возможностями могут использовать вычислительные ресурсы, которые еще несколько лет назад были исключительной прерогативой правительства, крупного бизнеса и исследовательских университетов, располагающих средствами для создания суперкомпьютеров.

- **Настольный компьютер бизнес-класса.** Руководство отделов информационных технологий коммерческих предприятий стремится обеспечить сотрудников вычислительными ресурсами, необходимыми для выполнения их работы, с наименьшими затратами. Большинству офисных работников не требуется исключительная графическая или вычислительная производитель-

ность, однако их компьютерные системы должны отвечать умеренным требованиям по воспроизведению видеопрезентаций для таких целей, как обучение.

Для бизнес-пользователей обычно более чем достаточно графического процессора, интегрированного с современным центральным процессором. За разумную цену бизнес-покупатели могут приобрести компьютерные системы с процессорами среднего уровня производительности с интегрированным графическим процессором. Подобные системы обеспечивают полную поддержку современных операционных систем и стандартных офисных приложений, таких как текстовый редактор, электронная почта и электронные таблицы. Если возникнет потребность в расширении возможностей системы по обработке графики, самый простой способ модернизации — установка карты графического процессора в разъем расширения.

- **Высокопроизводительный игровой компьютер.** Любителям новейших компьютерных 3D-игр требуется графический процессор с высочайшим уровнем производительности для отображения детализированных сцен с высоким разрешением при максимально возможной частоте кадров. Эти пользователи готовы вложить свои средства в мощный, дорогостоящий, требующий мощного питания графический процессор (или даже несколько процессоров) для достижения максимально возможной графической производительности.

  Не менее важной особенностью такого игрового компьютера, помимо мощного графического процессора, является наличие быстрого системного процессора. Системный и графический процессоры работают совместно, используя соединяющий их высокоскоростной интерфейс (обычно PCIe x16) для вычисления положения и направления обзора наблюдателя сцены, а также типа, местоположения, визуальных характеристик и ориентации всех объектов в сцене. Системный процессор передает эту геометрическую информацию графическому процессору, который выполняет математические операции, необходимые для создания реалистичного изображения на экране. Этот процесс повторяется с частотой, достаточной для плавного отображения сложных, быстро меняющихся сцен.

- **Смартфон высокого класса.** Современные смартфоны сочетают высокопроизводительные вычислительные возможности и высококачественные графические дисплеи с жесткими ограничениями по энергопотреблению и тепловыделению. Пользователи требуют быстрой, плавной и яркой графики для игр и просмотра видео, но при том они не согласны мириться с малым временем автономной работы или сильным нагревом устройства.

  Дисплеи современных телефонов содержат миллионы полноцветных пикселов, до 12 Гбайт оперативной памяти и до 1 Тбайт флеш-памяти. Эти устройства обычно оснащаются двумя камерами высокого разрешения (передней и задней), способными делать снимки и записывать видео. Телефоны высокого класса содержат 64-разрядный многоядерный процессор с интегрированным графическим процессором, а также множество функций, обес-

печивающих оптимальное сочетание энергоэффективности и высокой производительности.

Архитектуры смартфонов содержат процессоры DSP для решения таких задач, как кодирование и декодирование голосовых аудиосигналов во время телефонных разговоров и обработка принимаемых и передаваемых радиочастотных сигналов, используемых различными приемопередатчиками телефона. Типичный телефон поддерживает цифровую сотовую связь, Wi-Fi, Bluetooth и **ближнюю радиосвязь** (near-field communication, NFC). Современные смартфоны — это мощные вычислительные платформы, оснащенные широким спектром средств связи и оптимизированные для работы от аккумулятора.

В этом разделе мы рассмотрели архитектуры компьютерных систем, представляющие лишь малую часть современных и перспективных областей применения вычислительной техники. Независимо от назначения компьютерной системы, будь то настольный офисный компьютер, смартфон или система управления пассажирским авиалайнером, в процессе ее проектирования и реализации применяется единый набор общих архитектурных принципов.

Резюме

В этой главе было рассмотрено несколько специализированных областей вычислений, включая системы реального времени, цифровую обработку сигналов и обработку графической информации. После прочтения этой главы у вас должно сложиться более полное представление об особенностях современных компьютеров, включая работу в реальном времени, обработку аналоговых сигналов и графики в таких областях, как игры, голосовая связь, воспроизведение видео и использование графических процессоров в качестве миниатюрных суперкомпьютеров. Эти возможности являются важными расширениями основных вычислительных задач, выполняемых центральным процессором, будь то облачный сервер, настольный компьютер или смартфон.

В следующей главе мы обсудим современные процессорные архитектуры, в частности фон-неймановскую, гарвардскую и модифицированную гарвардскую. В ней будет рассмотрено использование виртуальной памяти со страничной организацией, а также возможности и функции модуля управления памятью.

Упражнения

1. **Частотно-монотонное планирование (RMS)** — это алгоритм назначения приоритетов потокам в приложениях жесткого реального времени с вытеснением, в которых потоки выполняются периодически.

   RMS назначает наивысший приоритет потоку с самым коротким периодом выполнения, следующий по значимости приоритет — потоку со следующим наи-

более коротким периодом выполнения и т. д. Система RMS является диспетчеризуемой, т. е. все ее задачи гарантированно укладываются в установленные сроки (при условии, что межпоточные взаимодействия или другие действия, такие как прерывания, не вызывают задержек обработки), если выполняется следующее условие:

$$\sum_{i=1}^{n}\frac{C_i}{T_i} \le n\left(2^{1/n}-1\right).$$

Эта формула выражает максимальную долю доступного времени обработки, которое может быть потрачено n потоками. В этой формуле C_i — это максимальное время выполнения, необходимое для потока i, а T_i — период выполнения потока i.

Является ли следующая система, состоящая из трех потоков, диспетчеризуемой?

Поток	Время выполнения C_i, мс	Период выполнения T_i, мс
Поток 1	50	100
Поток 2	100	500
Поток 3	120	1000

2. Широко используемая форма одномерного дискретного косинусного преобразования (ДКП) выражается следующей формулой:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right].$$

В этой формуле k, индекс коэффициента ДКП, меняется от 0 до $N-1$. Напишите программу для вычисления ДКП такой последовательности:

$$x = \{0.5, 0.2, 0.7, -0.6, 0.4, -0.2, 1.0, -0.3\}.$$

Косинусные члены в формуле зависят только от индексов n и k и не зависят от последовательности входных данных x. Это означает, что косинусные члены можно вычислить один раз и сохранить в виде констант для последующего использования. Если сделать это в качестве подготовительного этапа, то вычисление каждого коэффициента ДКП сводится к последовательности операций MAC.

Эта формула представляет собой неоптимизированную форму вычисления ДКП, требующую N^2 итераций операции MAC для вычисления всех коэффициентов ДКП в количестве N.

5. В качестве функции активации в ИНС часто используется гиперболический тангенс. Его функция определяется следующим образом:

$$\text{th}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Дан нейрон с входами от трех предшествующих нейронов, показанный на рис. 6.4. Вычислите выходной сигнал нейрона с гиперболическим тангенсом в качестве функции активации $F(x)$, используя следующие выходные сигналы предшествующих нейронов и веса соответствующих путей:

Нейрон	Выходной сигнал нейрона	Вес
N_1	0,6	0,4
N_2	–0,3	0,8
N_3	0,5	–0,2

7

Архитектура процессора и памяти

В этой главе более подробно рассматриваются архитектуры современных процессоров, в частности фон-неймановская, гарвардская и модифицированная гарвардская, а также области вычислений, в которых, как правило, применяется каждая из этих архитектур. Представлены концепции и преимущества виртуальной памяти со страничной организацией, широко используемой в вычислительных платформах потребительского и коммерческого назначения, а также в мобильных устройствах. Мы рассмотрим практические детали управления памятью на примере реальных систем — Windows NT и более поздних версий Windows. Глава завершается обсуждением особенностей и функций блока управления памятью.

После прочтения этой главы вы получите представление о ключевых особенностях архитектур современных процессоров и использовании физической и виртуальной памяти. Вы также ознакомитесь с преимуществами страничной организации памяти и функциями блока управления памятью.

В этой главе рассматриваются следующие темы:

- фон-неймановская, гарвардская и модифицированная гарвардская архитектуры;
- физическая и виртуальная память;
- виртуальная память со страничной организацией;
- блок управления памятью.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Фон-неймановская, гарвардская и модифицированная гарвардская архитектуры

В предыдущих главах мы кратко коснулись истории и современных областей применения процессорных архитектур — фон-неймановской, гарвардской и модифицированной гарвардской. В этом разделе мы рассмотрим эти конфигурации более подробно и изучим области вычислений, в которых обычно применяется каждая из указанных архитектур.

Фон-неймановская архитектура

Архитектура, получившая название фон-неймановской, была представлена Джоном фон Нейманом в 1945 г. Эта процессорная конфигурация состоит из устройства управления, арифметико-логического устройства (АЛУ), набора регистров и области памяти, содержащей программные инструкции и данные. Ключевая особенность, отличающая фон-неймановскую архитектуру от гарвардской, заключается в использовании единой области памяти для программных инструкций и данных, с которыми работают инструкции. Программистам принципиально просто, а разработчикам схем относительно проще разместить весь код и все данные, необходимые программе, в одной области памяти.

На рис. 7.1 показаны элементы фон-неймановской архитектуры.

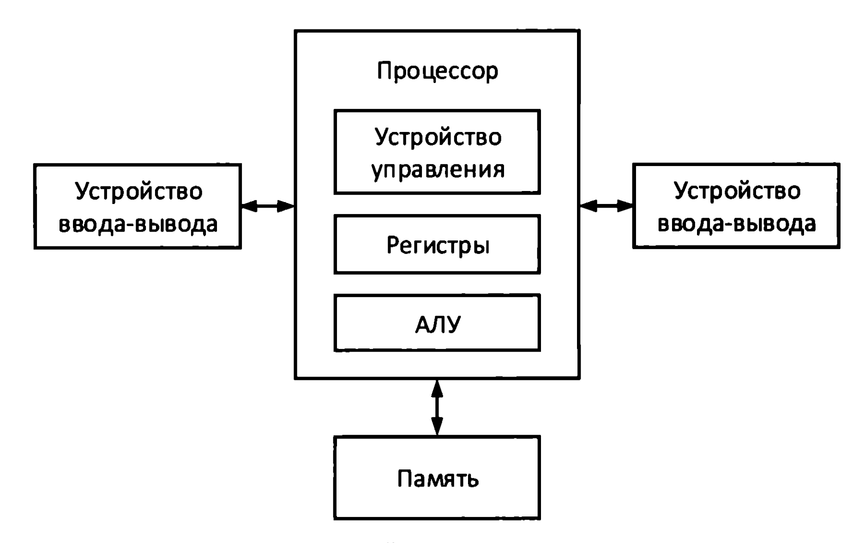


Рис. 7.1. *Фон-неймановская архитектура*

Архитектурный подход с единой памятью упростил проектирование и изготовление процессоров и компьютеров первых поколений, однако использование общей памяти для программ и данных создало определенные проблемы, связанные с производительностью систем и, в последние годы, с их безопасностью.

Вот некоторые из наиболее существенных проблем.

- **Узкое место архитектуры фон Неймана.** Использование единого интерфейса между процессором и основной памятью для доступа к инструкциям и данным часто требует нескольких циклов памяти для извлечения инструкции процессора и доступа к требуемым данным. Если непосредственное значение хранится рядом с опкодом инструкции, снижение производительности из-за узкого места может быть незначительным или вообще отсутствовать, поскольку, по крайней мере в некоторых случаях, непосредственное значение загружается вместе с опкодом за один цикл доступа к памяти. Однако большинство программ тратят значительную часть своего времени на работу с данными, хранящимися в выделенной области памяти отдельно от программных инструкций. В такой ситуации для извлечения опкода и любых необходимых элементов данных требуется несколько операций доступа к памяти.

  Использование кеш-памяти для программных инструкций и данных, которое подробно рассматривается в *главе 8*, может значительно смягчить это ограничение. Однако при работе с последовательностями инструкций и объектами данных, размер которых превышает объем кеш-памяти, преимущество кеширования уменьшается и, вероятно, даже на значительную величину. Невозможно избежать того факта, что размещение кода и данных в одной и той же области памяти с общим каналом обмена данными с процессором иногда будет ограничивать производительность системы.

- **Аспекты безопасности фон-неймановской архитектуры.** Использование единой области памяти для кода и данных открывает творческим программистам возможность хранить последовательности инструкций в памяти под видом данных, а затем отправлять эти инструкции в процессор на выполнение. Программы, которые записывают код в память, а затем выполняют его, реализуют **самомодифицирующийся код**. Помимо сложности поиска и устранения ошибок (поскольку многие средства отладки программного обеспечения основаны на предположении, что программа в памяти содержит все инструкции, которые были изначально собраны в нее при компиляции), эта возможность годами использовалась хакерами в злонамеренных целях.

- **Переполнение буфера** — удручающе распространенный недостаток в широко используемых программных решениях, таких как операционные системы, веб-серверы и базы данных. Переполнение буфера происходит, когда программа запрашивает входные данные и сохраняет их в буфере данных фиксированной длины. Если в коде не предусмотрена проверка длины вводимых данных, то пользователь может ввести последовательность, длина которой превышает доступное пространство для хранения. Когда это происходит, дополнительные данные перезаписывают содержимое ячеек памяти, предназначенных для других целей.

  Если перезаписываемый буфер хранится в стеке программы, то изобретательный пользователь может ввести длинную последовательность, способ-

ную перезаписать адрес возврата выполняемой в данный момент функции, который может храниться в том же стеке.

Тщательно продумав содержимое вводимой последовательности, злоумышленник может перехватить управление исполняемым приложением и направить его на выполнение любой желаемой последовательности инструкций. Для этого хакер должен подготовить последовательность, которая переполняет буфер ввода, перезаписывает адрес возврата функции другим, тщательно подобранным адресом и записывает в память последовательность инструкций, выполнение которой запускается по этому адресу. Последовательность инструкций, введенная злоумышленником, начинает выполняться, когда функция, которая первоначально запрашивала пользовательский ввод, осуществляет возврат, передавая управление коду хакера. Это позволяет хакеру "завладеть" компьютером.

На протяжении многих лет различные попытки решить проблему переполнения буфера отнимали огромное количество времени у исследователей компьютерной безопасности, начиная с первого случая широкого распространения атаки такого типа в 1988 г. Изготовители процессоров и разработчики операционных систем внедрили множество функций для борьбы с атаками переполнения буфера. Среди них решения с такими названиями, как **предотвращение выполнения данных** (data execution prevention, DEP) и **случайное распределение адресного пространства** (address space layout randomization, ASLR). Эти исправления продемонстрировали некоторую степень эффективности, однако фундаментальной особенностью процессора, которая создает возможность злоупотреблений такого типа, является использование одной и той же области памяти для программных инструкций и данных в фон-неймановской архитектуре.

Гарвардская архитектура

Гарвардская архитектура была первоначально реализована в компьютере Harvard Mark I в 1944 г. В строгой гарвардской архитектуре для программных инструкций и для данных отведены отдельные адресные пространства и шины доступа к памяти. Непосредственным преимуществом такой конфигурации является возможность одновременного доступа к инструкциям и к данным, что позволяет реализовать определенную форму параллелизма. Конечно, это улучшение достигается за счет существенного дублирования количества адресных строк, линий передачи данных и управляющих сигналов, которые должны быть реализованы в процессоре для доступа к обеим областям памяти.

На рис. 7.2 показана схема процессора, реализующего гарвардскую архитектуру.

Гарвардская архитектура может обеспечить более высокий уровень производительности за счет распараллеливания доступа к инструкциям и данным. Эта архитектура также устраняет весь класс проблем безопасности, связанных со злонамеренным выполнением программных инструкций, скрытых под видом данных, при условии, что память инструкций не может быть изменена программными инструкциями.

Данный подход предполагает, что память программ загружена инструкциями надлежащим образом.

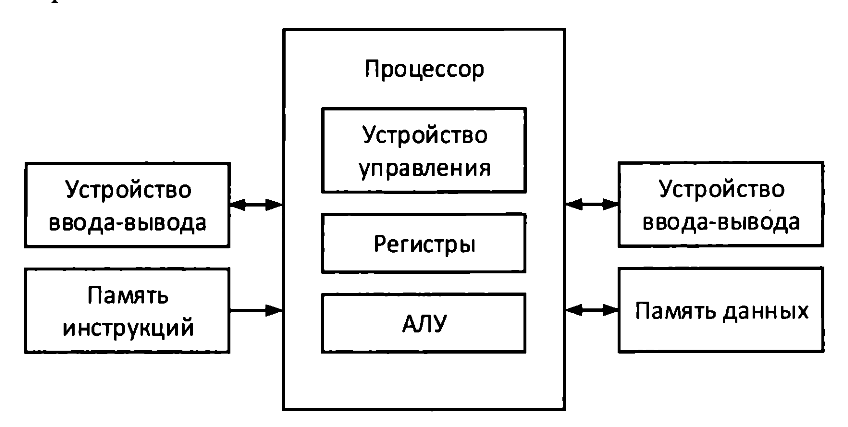


Рис. 7.2. *Гарвардская архитектура*

Если оглянуться назад, располагая знанием о распространении угроз безопасности, основанных на особенностях архитектуры фон Неймана, то есть повод задуматься, не было бы значительно лучше для всей отрасли информационных технологий, если бы еще на ранних этапах было достигнуто соглашение о внедрении гарвардской архитектуры с полным разделением областей памяти кода и данных, несмотря на связанные с этим затраты.

На практике строгая гарвардская архитектура редко применяется в современных компьютерах. Обычно используется несколько вариантов гарвардской архитектуры, которые в совокупности называются **модифицированными гарвардскими архитектурами**. Эти архитектуры являются темой следующего раздела.

Модифицированная гарвардская архитектура

В компьютерах, разработанных на основе **модифицированной гарвардской архитектуры**, реализована, как правило, некоторая степень разделения между программными инструкциями и данными. Это снижает влияние узкого места архитектуры фон Неймана и позволяет устранить некоторые проблемы безопасности, о которых мы уже говорили. Однако разделение между инструкциями и данными редко бывает абсолютным. В системах с модифицированной гарвардской архитектурой используются отдельные области памяти программных инструкций и памяти данных, но эти процессоры обычно поддерживают некоторые средства для сохранения данных в памяти программ и, в отдельных случаях, для сохранения инструкций в памяти данных.

На рис. 7.3 показана модифицированная гарвардская архитектура, представляющая многие реальные компьютерные системы.

Как мы видели в предыдущей главе, **процессоры цифровой обработки сигналов** (digital signal processors, DSP) получают значительные преимущества от использования архитектуры, подобной гарвардской.

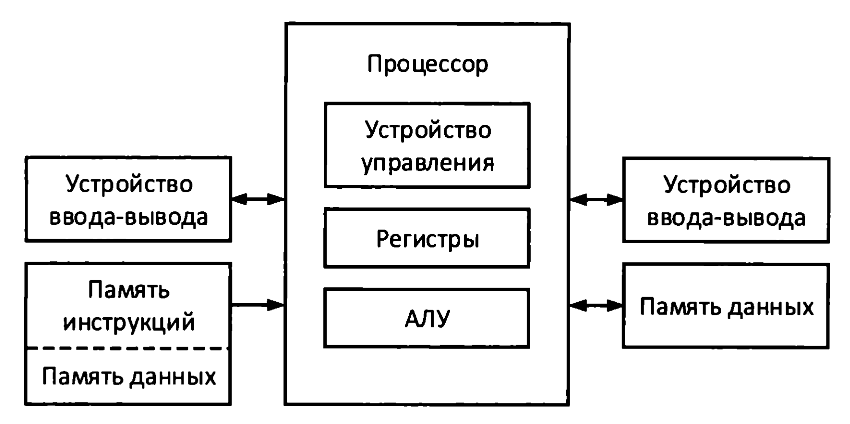


Рис. 7.3. Модифицированная гарвардская архитектура

Сохраняя один числовой вектор в памяти инструкций и второй вектор — в памяти данных, DSP может выполнить одну операцию **умножения с накоплением** (multiply accumulate, MAC) за один тактовый цикл процессора. В этих системах область памяти инструкций и содержащиеся в ней элементы данных обычно доступны только для чтения. На это указывает однонаправленная стрелка, соединяющая память инструкций с процессором на рис. 7.3. Поэтому для сохранения в области памяти инструкций подходят только значения постоянных данных.

Помимо DSP, большинство современных процессоров общего назначения содержат отдельные области кеш-памяти для инструкций и данных, реализуя тем самым важные особенности гарвардской архитектуры. Такие процессорные архитектуры, как *x*86 и ARM, поддерживают параллельный и независимый доступ к инструкциям и данным, когда запрашиваемые элементы находятся на первом уровне кеш-памяти. Если поиск во встроенной кеш-памяти не увенчался успехом, процессор должен обратиться к основной памяти по общей фон-неймановской шине, что занимает значительно больше времени.

На практике детали реализации конкретного процессора с точки зрения особенностей фон-неймановской и гарвардской архитектур редко имеют значение для разработчиков программного обеспечения, за исключением соображений производительности. Программисты обычно разрабатывают программы на выбранном ими языке высокого уровня, а детали операций, связанные с распределением данных и инструкций по соответствующим областям памяти, реализует компилятор или интерпретатор.

В следующем разделе обсуждаются преимущества виртуализации памяти.

Физическая и виртуальная память

Устройства памяти в компьютерах можно разделить на устройства **оперативной памяти** (оперативное запоминающее устройство, ОЗУ — random access memory, RAM), чтение из которых и запись в которые выполняются произвольно, и устройства **постоянной памяти** (постоянного запоминающего устройства, ПЗУ — read-

only memory, ROM), данные из которых, как следует из их названия, можно только считывать, но не записывать. Некоторые типы устройств памяти, такие как флеш-память и **электрически стираемое программируемое постоянное запоминающее устройство** (ЭСППЗУ), представляют золотую середину, когда содержимое устройств может быть изменено, но не так легко или не так быстро или не так много раз, как в случае обычной оперативной памяти.

Устройства памяти в компьютере должны быть настроены таким образом, чтобы гарантировать, что каждое устройство занимает уникальный участок системного адресного пространства, чтобы процессор мог получать доступ к каждому из (возможно) нескольких ОЗУ и ПЗУ путем выбора соответствующих адресных строк. Современные компьютерные системы обычно выполняют такое распределение адресного пространства автоматически, основываясь на разъеме, в который установлено устройство памяти.

Программное обеспечение, работающее на ранних компьютерных системах, а также на относительно несложных современных компьютерах и встраиваемых процессорах (таких как системы на базе 6502), для выполнения операций чтения и записи использует в программных инструкциях адреса в ОЗУ и ПЗУ.

Например, такая инструкция процессора 6502, как `JMP $1000`, предписывает процессору загрузить в указатель инструкций шестнадцатеричное значение `$1000` и выполнить инструкцию в этой ячейке памяти. При обработке этой инструкции устройство управления процессора 6502 помещает значение `$1000` на 16 адресных строк процессора и считывает байт по соответствующему адресу памяти. Этот байт интерпретируется как опкод следующей инструкции, которая должна быть выполнена. Аналогичным образом, загрузка байта из памяти с помощью такой инструкции, как `LDA $0200`, помещает значение `$0200` на адресные строки и копирует байт, хранящийся по этому адресу, в регистр A.

В системах, использующих физическую адресацию, адреса памяти в инструкциях являются фактическими адресами указанной инструкции или элемента данных. Это означает, что адрес памяти, содержащийся в инструкции, является тем же самым адресом, который используется для электрического доступа к соответствующей ячейке в устройстве памяти.

Этот архитектурный подход концептуально прост для реализации в процессорах, но в сценарии, включающем несколько одновременно выполняющихся программ (который называют **мультипрограммным режимом**), сложность разработки программного обеспечения может быстро стать чрезмерной. Если каждая из нескольких программ разрабатывается изолированно от других (например, в случае с несколькими независимыми разработчиками), должен быть какой-то способ распределения доступных адресных пространств ОЗУ и ПЗУ для отдельных программ, чтобы гарантировать одновременное выполнение нескольких программ (возможно, в контексте ОСРВ) без взаимных конфликтов при доступе к памяти.

Одной из хорошо известных ранних попыток поддержки выполнения нескольких программ в одном адресном пространстве на ПК является реализованная в операционной системе MS-DOS концепция **резидентной программы** (terminate and stay

resident, TSR). Резидентные программы резервируют область памяти и загружают в нее свой код, а затем возвращают управление операционной системе. Пользователи могут продолжать работать с системой в обычном режиме, загружая и используя другие приложения (по одному, конечно), но они также могут получить доступ к резидентной программе по мере необходимости, обычно с помощью специальной комбинации клавиш. Можно одновременно загрузить в память несколько резидентных программ, доступ к каждой из которых может осуществляться с помощью отдельной комбинации клавиш. После вызова резидентной программы пользователь взаимодействует с ней по мере необходимости, затем выполняет команду TSR для возвращения к запущенному в данный момент основному приложению.

Несмотря на ограничения во многих отношениях (в том числе потребление части оперативной памяти максимальным объемом 640 Кбайт, доступное на ранних ПК), резидентные программы фактически позволяли запускать несколько программ в едином адресном пространстве ОЗУ.

Разработка резидентных программ была сложной задачей. Более продвинутые резидентные программы, доступные в 1980-х и 1990-х годах, использовали недокументированные функции MS-DOS, чтобы обеспечить максимум возможностей для своих пользователей.

В результате таких сложностей резидентные программы приобрели репутацию вызывающих нестабильность системы. Было очевидно, что необходим другой подход к поддержке мультипрограммного режима.

Использование **виртуальной памяти** помогло преодолеть наиболее сложные проблемы, которые препятствовали широкому применению мультипрограммного режима в оригинальных ПК. Виртуальная память — это метод управления памятью, позволяющий каждому приложению работать в собственном пространстве памяти, внешне независимо от любых других приложений, которые могут одновременно находиться в запущенном состоянии в одной системе. На компьютере с системой управления виртуальной памятью операционная система отвечает за выделение физической памяти системным процессам и пользовательским приложениям. Аппаратные средства и программное обеспечение для управления памятью преобразуют запросы к памяти, исходящие из контекста виртуальной памяти приложения, в адреса физической памяти.

Помимо упрощения процесса разработки и запуска параллельных приложений, виртуальная память также позволяет выделять больший объем памяти, чем имеется в компьютере. Это достигается за счет использования вспомогательного хранилища (обычно файла на диске) для временного хранения копий разделов памяти, удаляемых из физической памяти, чтобы другая программа (или другая часть той же программы) могла запуститься в освободившейся памяти.

В современных компьютерах общего назначения разделы памяти обычно выделяются и перемещаются в количестве, кратном фрагменту фиксированного размера, называемому **страницей**. Страницы памяти, как правило, имеют размер 4 Кбайт или больше. Перемещение страниц памяти во вспомогательное хранилище и из не-

го в системах виртуальной памяти называется **подкачкой страниц**. Файл, содержащий замененные страницы, является **файлом подкачки**.

В системе с управлением виртуальной памятью ни разработчикам приложений, ни самому коду не нужно заботиться о том, сколько других приложений запущено в системе или насколько заполнена физическая память. По мере того, как приложение резервирует память для массивов данных и выполняет вызовы библиотечных подпрограмм (для чего требуется загрузка кода для этих подпрограмм в память), операционная система управляет распределением физической памяти и предпринимает необходимые действия для того, чтобы каждое приложение получало память по запросу. Только в маловероятном случае полного заполнения доступной физической памяти при одновременном достижении предельного размера файла подкачки система вынуждена возвращать код ошибки в ответ на запрос о выделении памяти.

Помимо упрощения работы программистов виртуальная память предоставляет ряд других важных преимуществ.

- Приложения могут игнорировать присутствие друг друга, а также не могут вмешиваться, случайно или намеренно, в работу друг друга. Аппаратные средства управления виртуальной памятью гарантируют, что каждое приложение получает доступ только к тем страницам памяти, которые были ему назначены.

  Попытки получить доступ к памяти другого процесса или к любому другому адресу за пределами выделенного приложению пространства памяти приводят к появлению исключения, вызванного **нарушением прав доступа**.
- Каждая страница памяти имеет набор атрибутов, которые ограничивают поддерживаемые ею типы операций. Страница может быть помечена как доступная только для чтения, что исключает любые попытки записи данных в нее. Страница может быть помечена как исполняемая — это означает, что она содержит код, который может быть выполнен в виде инструкций процессора. Страница может быть помечена как доступная для чтения и записи — такие страницы приложение может изменять по своему усмотрению. Устанавливая эти атрибуты надлежащим образом, операционные системы могут повысить стабильность системы, гарантируя невозможность внесения изменений в инструкции процессора и выполнения данных в виде инструкций независимо от того, является ли такая попытка результатом случайности или злого умысла.
- Страницам памяти может быть назначен лишь минимально необходимый уровень привилегий, что позволяет разрешить доступ к таким страницам только коду, запущенному с привилегиями ядра. Это ограничение гарантирует, что операционная система продолжит работать должным образом даже при наличии неправильно работающих приложений. Благодаря этому системную память можно отображать в адресное пространство каждого процесса, запрещая коду приложения напрямую взаимодействовать с этой памятью. Приложения могут обращаться к системной памяти только косвенно, через программный интерфейс, реализующий системные вызовы.

- Страницы памяти могут быть помечены для совместного использования приложениями. Это означает, что к такой странице может быть явно разрешен доступ более чем из одного процесса. Это обеспечивает эффективное взаимодействие между процессами.

В ранних версиях Microsoft Windows были реализованы некоторые функции виртуализации памяти с использованием возможностей сегментации памяти процессоров 80286 и 80386. В контексте Windows использование виртуальной памяти окончательно оформилось с выпуском Windows NT 3.1 в 1993 г. Архитектура системы Windows NT была основана на архитектуре **Virtual Address eXtension** (**VAX**), разработанной корпорацией Digital Equipment Corporation в 1970-х годах. В этой архитектуре была реализована 32-разрядная среда виртуальной памяти с виртуальным адресным пространством объемом 4 Гбайт, доступным для каждого из потенциально многих приложений, работающих в контексте мультипрограммного режима. Одним из основных архитекторов операционной системы VAX (**virtual memory system, VMS**), был Дэвид Катлер (David Cutler), который позже возглавил разработку Microsoft Windows NT.

Windows NT имеет плоскую 32-разрядную организацию памяти. Это означает возможность прямого доступа к любому адресу во всем 32-разрядном пространстве с использованием 32-разрядного адреса. Для манипулирования сегментными регистрами не требуется никаких дополнительных усилий программиста. По умолчанию виртуальное адресное пространство Windows NT разделено на два блока равного размера: пользовательское адресное пространство объемом 2 Гбайт в нижней половине диапазона и пространство ядра объемом 2 Гбайт в его верхней половине.

Следующий раздел посвящен виртуальной памяти со страничной организацией в 32-разрядной операционной системе Windows NT на процессорах Intel. Windows NT не в полной мере отражает реализацию виртуальной памяти в других операционных системах, однако в ней применены аналогичные принципы, а другие среды отличаются лишь в деталях. Это введение дает представление о концепциях виртуальной памяти, оставляя дополнительные подробности, связанные с более современными архитектурами, такими как 64-разрядные процессоры и операционные системы, для последующих глав.

Виртуальная память со страничной организацией

В 32-разрядной ОС Windows NT на процессорах Intel страницы памяти имеют размер 4 Кбайт. Это означает, что для адресации ячейки на конкретной странице требуется 12-битный адрес (т. к. $2^{12} = 4096$). Оставшиеся 20 бит 32-разрядного виртуального адреса используются в процессе преобразования (трансляции) виртуального адреса в физический.

В Windows NT все адреса памяти в программе (указанные как в исходном коде, так и в скомпилированном исполняемом коде) являются виртуальными адресами. Они не связаны с физическими адресами до тех пор, пока программа не будет запущена под управлением блока управления памятью.

Непрерывный раздел физической памяти размером 4 Кбайт в Windows NT называют **страничным кадром**. Это наименьшая единица памяти, управляемая системой виртуальной памяти Windows. Каждый страничный кадр начинается с границы в 4 Кбайт. Это означает, что в начале любого страничного кадра все младшие 12 бит адреса равны нулю. Система отслеживает информацию, связанную со страничными кадрами, с помощью таблиц страниц.

Размер **таблицы страниц** в Windows NT подобран таким образом, чтобы она занимала одну страницу размером 4 Кбайт. Каждая 4-байтовая запись в таблице страниц позволяет транслировать 32-разрядный адрес из виртуального адресного пространства, используемого программными структурами, в физический адрес, необходимый для доступа к ячейке памяти в ОЗУ или ПЗУ. Таблица страниц размером 4 Кбайт содержит 1024 записи трансляции адресов страниц. Одна таблица страниц управляет доступом к 4 Мбайт адресного пространства: умножаем 1024 страничных кадра каждой таблицы на 4 Кбайт для каждой страницы. Процесс может иметь несколько связанных с ним таблиц страниц, а управление всеми этими таблицами осуществляется с помощью каталога таблиц страниц.

Каталог таблиц страниц представляет собой страницу размером 4 Кбайт, содержащую серию 4-байтовых ссылок на таблицы страниц. Каталог таблиц страниц может содержать 1024 ссылки на таблицы страниц. Один каталог таблиц страниц охватывает все 4 Гбайт адресного пространства 32-разрядной ОС Windows NT (умножаем 4 Мбайт для каждой таблицы страниц на 1024 ссылки на таблицы страниц).

Каждый процесс Windows NT имеет отдельный каталог таблиц страниц, набор таблиц страниц и набор страничных кадров, выделенных для использования этим процессом. Таблицы страниц процесса применимы ко всем потокам процесса, поскольку все потоки процесса совместно используют одно и то же адресное пространство и распределение памяти.

Когда системный планировщик переключается с одного процесса на другой, контекст виртуальной памяти входящего процесса заменяет контекст исходящего процесса.

Процессоры Intel *x*86 поддерживают адрес текущего каталога таблиц страниц процесса в регистре `CR3`, также известном как **базовый регистр каталога страниц** (page directory base register, PDBR). Это точка входа в каталог таблиц страниц и таблицы страниц, с помощью которой процессор может преобразовать любой действительный виртуальный адрес в соответствующий физический адрес.

Обращаясь к произвольной (допустимой) ячейке памяти и предполагая, что информация, которая могла бы ускорить доступ, еще не сохранена в кеше недавних трансляций виртуального адреса в физический, процессор сначала ищет физический адрес соответствующей таблицы страниц в каталоге таблиц страниц, используя старшие 10 бит виртуального адреса. Затем он обращается к нужной таблице страниц и использует следующие по значению 10 бит адреса для выбора физической страницы, содержащей запрошенные данные. После этого младшие 12 бит адреса указывают в страничном кадре ячейку памяти, запрошенную исполняемой инструкцией.

СТРАНИЧНЫЕ КАДРЫ НЕ ОТРАЖАЮТ ДЕЙСТВИТЕЛЬНЫЕ РАЗДЕЛЫ ФИЗИЧЕСКОЙ ПАМЯТИ



Физическая память на самом деле не разделена на страничные кадры. Структура страницы — это лишь метод, используемый системой для отслеживания информации, необходимой для трансляции виртуальных адресов в адреса ячеек физической памяти.

Для того чтобы соответствовать ожиданиям пользователей в отношении производительности, каждое обращение к памяти должно быть как можно более быстрым. Во время выполнения каждой инструкции для извлечения кодов операций инструкций и данных выполняется по крайней мере одно преобразование виртуального адреса в физический. Из-за высокой частоты повторения этого процесса разработчики процессоров прилагают значительные усилия для достижения максимальной эффективности трансляции виртуальных адресов.

В современных процессорах в кеше трансляций сохраняются результаты недавних поисков для трансляции адресов виртуальной памяти. Такой подход позволяет выполнять весьма значительную долю трансляций адресов виртуальной памяти внутри процессора без дополнительных тактовых циклов, которые потребовались бы, если бы процессору необходимо было сначала осуществить поиск адреса таблицы страниц в каталоге таблиц страниц, а затем получить доступ к таблице страниц для определения запрошенного физического адреса.

Структуры данных, используемые при трансляции виртуального адреса в физический, недоступны приложениям, работающим на уровне привилегий пользователя. Все действия, связанные с трансляцией адресов, выполняются в аппаратных средствах процессора и в программных процессах, работающих в режиме ядра.

Для того чтобы прояснить процедуру трансляции виртуальных адресов, на рис. 7.4 представлен пример трансляции 32-разрядного виртуального адреса в физический в ОС Windows.

Рассмотрим процесс трансляции, представленный на рис. 7.4, шаг за шагом. Предположим, что процессор запрашивает 8-битное значение данных, хранящееся по виртуальному адресу `$00402003`, с помощью инструкции `moval,[ebx]`, при этом в регистр `ebx` ранее было загружено значение `$00402003`. Также предположим, что данные трансляции этого адреса еще не были сохранены в кеше недавних трансляций виртуального адреса в физический и страница находится в основной памяти. Процесс трансляции описывает следующая процедура.

1. Процессор пытается выполнить инструкцию `moval,[ebx]`, но он не может сделать это, поскольку у него нет непосредственного доступа к информации, необходимой для трансляции виртуального адреса, хранящегося в `ebx`, в соответствующий физический адрес. Эта ситуация порождает исключение **"отказ страницы"**, в результате управление передается операционной системе, чтобы она могла разрешить проблему с трансляцией адреса. В данном случае использова-

ние термина *"отказ"* не означает, что произошла какая-то ошибка. Отказы страниц — это обычная составляющая выполнения приложения.

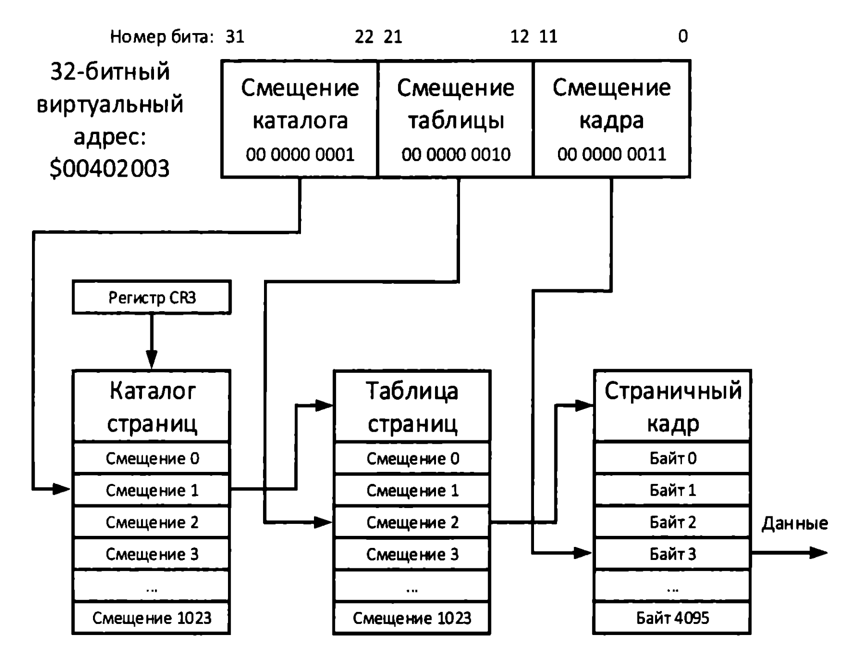


***Рис. 7.4.** Трансляция виртуального адреса в физический*

2. Запрошенный виртуальный адрес сдвигается вправо на 22 битовые позиции, оставляя 10-битное смещение каталога, которое в этом примере имеет значение 1.
3. Смещение каталога сдвигается влево на 2 битовые позиции (поскольку каждая запись в каталоге страниц составляет 4 байта) и добавляется к содержимому регистра CR3 процессора (базовый регистр каталога страниц). Результатом является адрес записи в каталоге таблиц страниц, содержащей адрес соответствующей таблицы страниц.
4. Запрошенный виртуальный адрес сдвигается вправо на 12 битовых позиций и маскируется, чтобы осталось только 10-битное смещение таблицы, которое в этом примере имеет значение 2.
5. Смещение таблицы сдвигается влево на 2 битовых позиции (поскольку каждая запись в этой таблице также имеет длину 4 байта) и добавляется к адресу таблицы страниц, указанному на *шаге 3*. 32-разрядный адрес, считываемый из этой ячейки, является физическим адресом кадра страницы, содержащего запрошенные данные.
6. Процессор сохраняет данные трансляции, которая представляет собой преобразование старших 20 битов виртуального адреса в соответствующие старшие 20 битов адреса страничного кадра, в своем кеше трансляций.
7. Процессор снова пытается выполнить инструкцию `mov al,[ebx]`, которая теперь успешно перемещает запрошенный байт данных в регистр al, используя поме-

щенные в кеш данные трансляции виртуального адреса в физический. Для доступа к запрошенному байту младшие 12 бит виртуального адреса (смещение кадра), содержащие значение 3 в данном примере, добавляются к адресу кадра страницы, вычисленному на *шаге 5*.

После завершения этих действий трансляция для запрошенного страничного кадра в течение некоторого времени остается доступной в кеш-памяти трансляций процессора. Пока данные трансляции остаются в кеше, последующие запросы того же виртуального адреса или других ячеек в том же страничном кадре будут выполняться без задержки до тех пор, пока запись кеша для этого страничного кадра не будет перезаписана в ходе последующего выполнения кода.

Отказ страницы, процедура обработки которого описана на предыдущих шагах, называется **мягким отказом страницы**. Он запускает трансляцию виртуального адреса в физический для страницы, которая уже доступна процессору, но отсутствует в кеше трансляций.

Жесткий отказ страницы возникает при попытке доступа к странице, которая была перенесена во вспомогательное хранилище. Обработка жесткого отказа страницы требует нескольких дополнительных шагов, включая резервирование страничного кадра для получения запрошенного кадра, запрос страницы из вспомогательного хранилища и обновление физического адреса страницы в таблице страниц. Поскольку жесткие отказы страниц связаны с дисковыми операциями, отказы этого типа оказывают гораздо большее влияние на производительность приложения, чем мягкие отказы страниц.

Процесс трансляции переводит старшие 20 бит виртуального адреса в соответствующие 20 бит физического адреса. При этом 12 бит в каждой записи таблицы страниц остаются доступными для сохранения информации о состоянии и конфигурации для страничного кадра. Использование этих битов описано в следующем разделе.

Биты состояния страницы

В табл. 7.1 описан каждый из 12 бит состояния в 32-разрядной записи таблицы страниц Windows NT.

Таблица 7.1. *Биты состояния страницы*

Бит	Название	Описание
0	Valid (Допустимость трансляции)	Значение 1 указывает, что эта запись таблицы страниц может быть использована для трансляции. Если этот бит равен 0, остальные биты могут иметь разные значения, как определено в операционной системе. В описаниях следующих битов предполагается, что бит Valid равен 1
1	Write (Запись)	Значение 1 указывает, что страница доступна для записи, значение 0 — страница доступна только для чтения

Таблица 7.1 (окончание)

Бит	Название	Описание
2	Owner (Владелец)	Значение 1 указывает, что страница находится в режиме пользователя, значение 0 — страница находится в режиме ядра
3	Write through (Сквозная запись)	Значение 1 указывает, что изменения на странице должны быть немедленно сохранены на диск, значение 0 — изменения страницы будут сохраняться в ОЗУ
4	Cache disabled (Кеш отключен)	Значение 1 указывает, что кеширование данной страницы отключено, значение 0 — кеширование включено
5	Accessed (Факт доступа)	Значение 1 указывает, что страница была прочитана или записана, значение 0 — доступ к странице (любым способом) не осуществлялся
6	Dirty (Факт записи)	Значение 1 указывает, что страница была записана, значение 0 — запись не производилась
7	Reserved (Резерв)	Не используется
8	Global (Глобально)	Значение 1 указывает, что это преобразование применимо ко всем процессам, значение 0 — эта запись применима только к одному процессу
9	Reserved (Резерв)	Не используется
10	Reserved (Резерв)	Не используется
11	Reserved (Резерв)	Не используется

Процессор использует биты состояния страницы для сохранения информации о содержимом страницы и для управления доступом к каждой странице со стороны системных и пользовательских процессов. Бит Owner (Владелец) идентифицирует страницу как принадлежащую ядру или пользователю. Пользовательские процессы не могут читать или записывать какие-либо страницы, принадлежащие ядру. Любая попытка записи на страницу, которая помечена как доступная только для чтения (бит Write (Запись) равен 0), приводит к исключению нарушения доступа.

Система использует биты состояния страницы для максимально эффективного управления памятью. Если бит Accessed (Факт доступа) имеет нулевое значение, страница была зарезервирована, но никогда не использовалась. Когда системе необходимо освободить физическую память, страницы, к которым никогда не обращались, являются основными кандидатами на удаление, поскольку при удалении из памяти нет необходимости сохранять их содержимое. Аналогично, если бит Dirty (Факт записи) имеет нулевое значение, данные страницы не менялись с тех пор, как они были перенесены в память. Диспетчер памяти может освобождать страницы с нулевым битом Dirty (Факт записи), поскольку известно, что когда страница снова

понадобится, ее можно будет опять загрузить из исходного местоположения (обычно это файл на диске), чтобы в точности восстановить.

Страницы с установленным битом Dirty (Факт записи) должны быть сохранены в файле подкачки при их удалении из памяти. Когда страница перемещается в файл подкачки, соответствующая запись в таблице страниц обновляется с использованием другого формата из табл. 7.1, чтобы указать на недопустимость использования этой страницы для преобразования (бит Valid (Допустимость трансляции) равен 0) и сохранить его местоположение в пределах файла подкачки.

Формат записей таблицы страниц определяется архитектурой процессора, которой в данном случае является архитектура семейства Intel *x*86.

Аппаратные средства процессора обращаются к записям таблицы страниц для трансляции виртуального адреса в физический и для обеспечения защиты страниц при работе процессора на полной скорости.

В дополнение к управлению памятью, используемой каждым процессом, система должна отслеживать все страничные кадры ОЗУ и ПЗУ на компьютере независимо от того, используются они процессом или нет. Система сохраняет эту информацию в списках, называемых **пулами памяти**, которые рассматриваются далее.

Пулы памяти

В Windows NT пулы памяти разделены на два типа: с подкачкой и без нее.

- **Пул без подкачки** содержит все страничные кадры, которые гарантированно всегда остаются резидентными в памяти. Код для процедур обслуживания прерываний, драйверов устройств и самого диспетчера памяти всегда должен оставаться немедленно доступным для процессора по соображениям поддержания требуемой производительности системы, а в случае самого диспетчера памяти — для того, чтобы система вообще могла функционировать. Невыгружаемые виртуальные адреса находятся в системной части виртуального адресного пространства процесса.
- **Пул с подкачкой** содержит страницы виртуальной памяти, которые при необходимости могут быть временно выгружены из физической памяти.

Система отслеживает состояние кадра каждой страницы в физической памяти с помощью структуры, называемой базой данных **номеров страничных кадров** (page frame number, PFN). PFN — это старшие 20 бит физического базового адреса страничного кадра. Кадр каждой страницы может находиться в одном из нескольких состояний в зависимости от его текущего и предыдущего использования. Вот некоторые из основных состояний страничного кадра.

- **Active** (Активный). Страничный кадр является частью рабочего набора системного или пользовательского процесса. **Рабочий набор** — это часть виртуального адресного пространства процесса, присутствующая в данный момент в физической памяти.

- **Standby** (Ожидающий). Ожидающие страницы — это страницы, которые были удалены из рабочих наборов процесса и не были изменены.
- **Modified** (Измененный). Измененные страницы были удалены из рабочих наборов процесса и были изменены. Эти страницы должны быть записаны на диск, прежде чем их кадры могут быть использованы повторно.
- **Free** (Свободный). Свободные страницы не используются, но еще содержат данные о своем последнем участии в рабочем наборе. По соображениям безопасности эти страницы не могут быть доступны пользовательскому процессу до тех пор, пока их содержимое не будет заменено нулями.
- **Zeroed** (Обнуленный). Обнуленные страницы являются свободными, а их содержимое было перезаписано нулями. Эти страницы доступны для резервирования пользовательскими процессами.
- **Bad** (Поврежденный). Поврежденные страницы вызвали аппаратные ошибки при обращении к ним процессора. Такие страницы отслеживаются в базе данных PFN и не используются операционной системой.

По мере запуска, выполнения и завершения работы системных служб и приложений страничные кадры переходят из одного состояния в другое под управлением системы. В Windows NT системная задача выполняется в периоды простоя и преобразует свободные страницы в обнуленные, перезаписывая эти страницы нулями.

Обсуждение в этом разделе было сосредоточено на реализации виртуальной памяти в архитектуре процессоров *x*86 под управлением ОС Windows NT. Другие процессорные архитектуры и операционные системы реализуют виртуальную память, используя аналогичные концепции.

Компонент процессора, управляющий распределением памяти, преобразованием адресов и функциями защиты, называется **блоком управления памятью**. В следующем разделе он рассматривается как обобщенный компонент компьютерной системы.

Блок управления памятью

В процессорных архитектурах, поддерживающих виртуальную память со страничной организацией, функции **блока управления памятью** (memory management unit, MMU) реализуются либо внутри самого процессора, либо иногда (особенно в достаточно старых решениях) в виде отдельной интегральной схемы. В MMU виртуальное адресное пространство процессора разделено на единицы распределения памяти размером со страницу.

Страницы могут иметь фиксированный размер, как в случае Windows NT, либо MMU может поддерживать несколько размеров. Современные процессоры, включая процессоры *x*86 более поздних поколений, часто поддерживают страницы двух размеров: малые и большие. Малые страницы обычно имеют размер несколько килобайтов, а большая страница может занимать несколько мегабайтов. Поддержка

больших страниц позволяет избежать неэффективности, связанной с выделением множества страниц меньшего размера при работе с крупными объектами данных.

Как обсуждалось ранее, MMU обычно содержит кеш для повышения скорости доступа к памяти за счет исключения необходимости просматривать каталог таблиц страниц и выполнять поиск в таблице страниц при каждом обращении к памяти. Использование кеширования для повышения производительности является темой *главы 8*, однако здесь мы представим кеш для трансляции виртуальных адресов в физические, поскольку эта конструкция является основной особенностью большинства архитектур MMU.

Компонент кеширования в MMU, который хранит данные ранее выполненных трансляций виртуальных адресов в физические, называется **буфером ассоциативной трансляции** (translation lookaside buffer, TLB). Для того чтобы избежать поиска таблицы страниц в каталоге таблиц страниц, а затем поиска в найденной таблице страниц кадра нужной страницы при каждом обращении к памяти, TLB сохраняет данные о трансляции виртуальных адресов в страничные кадры, полученные в результате этих поисков, в аппаратной структуре, называемой **ассоциативной памятью**.

Каждый раз, когда процессору требуется получить доступ к физической памяти, что может происходить несколько раз за время выполнения одной инструкции, он сначала проверяет ассоциативную память TLB, чтобы определить, нет ли в TLB информации о нужной трансляции. Если такая информация в буфере имеется, инструкция немедленно использует ее для доступа к физической памяти. Если TLB не содержит данных запрошенной трансляции, возникает отказ страницы, и процессор должен заново просмотреть каталог таблиц страниц и таблицу страниц, чтобы определить транслированный адрес, предполагая, что нужная страница находится в памяти.

На рис. 7.5 представлена работа буфера TLB.

Буфер ассоциативной трансляции (TLB)

Одновременные сравнения

Виртуальная страница $10000

Номер виртуальной страницы	Номер страничного кадра
Виртуальная страница $38533	Страничный кадр $50643
Виртуальная страница $21BB7	Страничный кадр $ABE4D
Виртуальная страница $5D633	Страничный кадр $46E59
Виртуальная страница $12676	Страничный кадр $178D8
...	...
Виртуальная страница $82F3B	Страничный кадр $03D52
Виртуальная страница $10000	Страничный кадр $55FAE
Виртуальная страница $03CD3	Страничный кадр $6E821

Совпадение

Рис. 7.5. *Работа буфера ассоциативной трансляции*

При каждом обращении к памяти процессор извлекает из виртуального адреса старшие 20 бит, чтобы идентифицировать номер виртуальной страницы. Этот номер страницы, `$10000`, в данном примере используется для поиска в TLB соответствующей записи. Аппаратные средства TLB одновременно сравнивают номер запрошенной виртуальной страницы с номерами всех виртуальных страниц, хранящихся в TLB. Если найдено совпадение, номер кадра соответствующей страницы немедленно предоставляется для получения доступа к физической памяти.

TLB содержит ограниченное количество записей, обычно 64 или меньше. Процессор должен управлять тем, какие записи TLB сохраняются, а какие отбрасываются по мере обработки запросов адресов для разных ячеек памяти. Когда буфер TLB заполнен, MMU решает, какую из существующих записей TLB перезаписать новой информацией. MMU может случайным образом выбрать запись TLB для замены, более сложная реализация MMU может использовать информацию о сроке существования записей для замены записи TLB, которая дольше всех не использовалась.

В дополнение к трансляции виртуального адреса в физический MMU обычно выполняют следующие функции.

- **Разделение виртуальной памяти на пространство ядра и пространство пользователя.** Память ядра зарезервирована для использования операционной системой и связанными с ней компонентами, такими как драйверы устройств. Пространство пользователя доступно для приложений и для действий, инициируемых пользователями, например для обработки команд, набранных в окне командной строки. Код пользовательского уровня не может получить прямой доступ к системной памяти. Вместо этого код пользователя должен вызывать системные функции для запроса таких услуг, как выделение памяти и операции ввода-вывода.
- **Изоляция областей памяти процессов.** Каждый процесс имеет собственное адресное пространство, являющееся единственной областью памяти, к которой ему разрешен доступ. Если между процессами не установлена разрешенная системой область памяти совместного использования, любому процессу запрещено обращаться к памяти, используемой другим процессом. Один процесс не может ошибочно или намеренно изменить содержимое памяти, которая относится к частной области памяти для другого процесса.
- **Ограничения доступа на уровне страниц.** В дополнение к защите системных страниц от доступа пользователей и защите частных страниц процессов от доступа других процессов процесс может устанавливать защиту на отдельные страницы, которыми он владеет. Страницы могут быть помечены как доступные только для чтения, что запрещает изменение их содержимого. Страницы, помеченные как запрещенные для исполнения, не могут быть использованы для предоставления инструкций, выполняемых процессором. В некоторых архитектурах страницы могут быть помечены как запрещенные для доступа, что устанавливает запрет как на чтение, так и на запись. Страницы, содержащие исполняемый код, могут быть помечены как доступные

только для чтения, чтобы предотвратить случайное или намеренное изменение инструкций в памяти.

- **Обнаружение проблем с программным обеспечением.** В некоторых языках программирования, в частности в языке C, к сожалению, часто наблюдаются попытки использовать указатель, содержащий недопустимый адрес (**указатель** — это переменная, содержащая адрес другой переменной). Наиболее часто в такой ситуации встречается недопустимый адрес 0, поскольку переменные нередко инициализируются нулем. Эта проблема настолько распространена, что реакция системы на нее имеет собственное название — **исключение нулевого указателя**. Когда программа на языке C пытается получить доступ к ячейке памяти, которая не входит в допустимый диапазон виртуальных адресов программы, например к адресу `$00000000`, MMU вызывает исключение, которое, если его не обработать в программе, обычно приводит к аварийному завершению программы с выводом сообщения об ошибке в окно консоли. В системах без виртуальной памяти обращения к ошибочным ячейкам могут просто приводить к считыванию или записи памяти по указанному адресу без какой-либо индикации ошибки, что ведет к некорректной работе приложения или всей системы. Такие ошибки в системах без MMU могут быть крайне сложными для исправления, если проблема проявляется не сразу.

Современные процессоры под управлением Linux, Windows и большинства операционных систем для мобильных устройств обычно требуют, чтобы их хост-системы использовали управление виртуальной памятью и обеспечивали механизмы защиты страниц, описанные в этой главе.

Встраиваемые процессоры реального времени, выполняющие критически важные для безопасности задачи, такие как управление полетом авиалайнера или работой автомобильной подушки безопасности, могут поддерживать или не поддерживать полный набор функций MMU. Одним из недостатков, связанных с использованием виртуальной памяти в системах жесткого реального времени, является переменная временна́я задержка, возникающая в результате необходимости обработки мягких отказов страниц, а также, если реализована подкачка страниц, то и жестких отказов страниц. Поскольку во многих ОСРВ время выполнения должно строго контролироваться, их разработчики часто избегают использования виртуальной памяти. Такие системы не содержат MMU, но в них часто реализуются многие иные возможности MMU, такие как аппаратная защита системной памяти и управление доступом к областям оперативной памяти.

Резюме

В этой главе были рассмотрены основные категории архитектур современных процессоров, включая фон-неймановскую, гарвардскую и модифицированную гарвардскую, а также их использование в различных областях вычислений. Были обсуждены концепции виртуальной памяти со страничной организацией, включая

некоторые детали, относящиеся к реализации такой памяти в Windows NT на процессоре *x*86.

Рассмотрена общая структура блока управления памятью (MMU) с упором на использование буфера ассоциативной трансляции (TLB) в качестве метода оптимизации производительности при трансляции виртуальных адресов в физические.

В следующей главе мы не ограничимся повышением производительности, обеспечиваемым TLB, и подробно рассмотрим широко используемые методы ускорения процессоров, включая кеширование, конвейерную обработку инструкций и параллелизм инструкций.

Упражнения

1. 16-разрядный встраиваемый процессор имеет отдельные области памяти для кода и данных. Код хранится во флеш-памяти, а изменяемые данные — в оперативной памяти. Некоторые значения данных, такие как константы и начальные значения для элементов данных оперативной памяти, хранятся в той же области флеш-памяти, что и инструкции программы. ОЗУ и ПЗУ находятся в одном адресном пространстве. Какая из архитектур процессоров, рассмотренных в данной главе, лучше всего описывает этот процессор?
2. Процессор, описанный в *упражнении 1*, имеет функции защиты памяти, которые не позволяют исполняемому коду изменять память инструкций программы. Для доступа к инструкциям и данным этот процессор использует физические адреса. Содержит ли он блок управления памятью (MMU)?
3. Порядок доступа к последовательным элементам в большой структуре данных может оказать ощутимое влияние на скорость обработки из-за таких факторов, как повторное использование записей буфера TLB. Последовательный доступ к отдаленным элементам массива (это элементы, находящиеся за пределами страничного кадра, где размещены ранее запрошенные элементы) требует частых мягких отказов страниц по мере загрузки новых записей TLB и удаления старых.

   Напишите программу, которая создает большой двумерный массив чисел, например на 10 000 строк и 10 000 столбцов. Выполните итеративный обход массива в порядке возрастания столбцов, подписывая каждый элемент суммой индексов строки и столбца. Доступ по столбцам означает, что индекс столбца (второй индекс) увеличивается быстрее всего. Другими словами, индекс столбца увеличивается во внутреннем цикле.

   Точно измерьте время, которое занимает эта процедура. Обратите внимание, что вам может потребоваться принять меры для того, чтобы ваш язык программирования не исключил при оптимизации весь этот расчет из-за того, что результаты работы с массивом не используются в дальнейшем. Может быть достаточно вывести одно из значений массива после завершения отсчета времени или же может потребоваться сделать что-то вроде суммирования всех элементов массива и вывода этого результата.

Повторите процесс, включая измерение времени, точно так же, как было описано ранее, только измените внутренний цикл на итерацию по индексу строки (первый индекс), а внешний цикл — на итерацию по индексу столбца, сделав последовательность доступа построчной.

Во время выполнения вашего кода компьютеры общего назначения выполняют множество других задач, поэтому, чтобы получить статистически достоверный результат, вам может понадобиться выполнить обе процедуры несколько раз. Для начала можно провести эксперимент 10 раз и усреднить время доступа к массиву по столбцам и по строкам.

Можете ли вы определить метод доступа к массиву, постоянно дающий лучший результат? Какой порядок является самым быстрым в вашей системе при использовании выбранного вами языка? Следует учитывать, что разница между методами с порядком доступа по столбцам и по строкам может быть не слишком значительной — она может составлять всего несколько процентов.



Методы повышения производительности

Фундаментальные аспекты архитектур процессоров и памяти, рассмотренные в предыдущих главах, служат основой для полнофункциональной компьютерной системы. Однако производительность такой системы по сравнению с большинством современных процессоров будет невысокой, если в нее не добавить функции, позволяющие увеличить скорость выполнения инструкций.

При проектировании процессоров и систем обычно используют несколько методов повышения производительности для достижения максимальной скорости выполнения в реальных компьютерных системах. Эти методы не меняют то, что делает процессор с точки зрения выполнения программ и обработки данных; они просто помогают делать это быстрее.

После прочтения этой главы вы поймете выгоду от применения многоуровневой кеш-памяти в компьютерных архитектурах, а также будете ориентироваться в преимуществах и проблемах, связанных с конвейерной обработкой инструкций. Вы также узнаете, как повысить производительность с помощью одновременной многопоточности, и ознакомитесь с назначением и применением модели обработки с одним потоком инструкций и множеством потоков данных (single instruction, multiple data, SIMD).

В этой главе будут рассмотрены следующие темы:

- кеш-память;
- конвейерная обработка инструкций;
- одновременная многопоточность;
- модель обработки SIMD.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Кеш-память

Кеш-память — это область высокоскоростной памяти (по сравнению со скоростью основной памяти), в которой временно хранятся программные инструкции или данные для будущего использования. Как правило, эти инструкции или элементы данных были недавно извлечены из основной памяти и, вероятно, вскоре снова понадобятся.

Основное назначение кеш-памяти — увеличить скорость повторного доступа к одной и той же ячейке памяти, а также к ячейкам памяти, расположенным рядом с ней. Для того чтобы обеспечить эффективное ускорение, доступ к кешированным элементам должен быть значительно быстрее, чем доступ к исходному источнику инструкций или данных, называемому **резервным хранилищем**.

Когда используется кеширование, каждая попытка доступа к ячейке памяти начинается с поиска в кеше. Если запрошенный элемент присутствует в кеше, процессор немедленно извлекает его и использует. Это называется **кеш-попаданием**. Если поиск в кеше не увенчался успехом (**кеш-промах**), инструкция или элемент данных извлекается из резервного хранилища. В процессе извлечения запрошенного элемента его копия добавляется в кеш для предполагаемого использования в будущем.

Кеш-память используется в компьютерных системах для различных целей. Ниже приведены некоторые примеры применения кеш-памяти.

- **Буфер ассоциативной трансляции (translation lookaside buffer, TLB).** TLB, как было показано в *главе 7*, представляет собой вариант кеш-памяти, которая используется в процессорах, поддерживающих виртуальную память со страничной организацией. TLB содержит набор записей о трансляции виртуальных адресов в физические, которые ускоряют доступ к страничным кадрам в физической памяти. По мере выполнения инструкций каждое обращение к основной памяти требует трансляции виртуального адреса в физический. Успешный поиск в TLB приводит к гораздо более быстрому выполнению инструкций по сравнению с процессом поиска в таблицах страниц после промаха TLB. Буфер TLB является частью блока управления памятью (MMU) и не имеет прямого отношения к вариантам кеш-памяти процессора, обсуждаемым далее в этом разделе.
- **Кеш-память дисковых накопителей.** Операции чтения и записи на намагниченные пластины дисковых накопителей выполняются на порядки медленнее, чем доступ к устройствам **динамической памяти произвольного дос-**

тупа (dynamic RAM, DRAM). В дисковых накопителях кеш-память обычно используется для хранения выходных данных операций чтения и временного хранения данных при подготовке к записи. Контроллеры накопителей часто хранят во внутренней кеш-памяти больше данных, чем первоначально запрошенное количество, в расчете на то, что при будущих обращениях будут запрашиваться данные, хранящиеся рядом с первоначально запрошенными.

Если это предположение окажется правильным, как часто бывает, накопитель может немедленно отреагировать на второй запрос, передав данные из кеша без задержки, связанной с доступом к дискам.

- **Кеш-память веб-браузеров.** Веб-браузеры обычно хранят копии недавно посещенных веб-страниц в памяти в ожидании нажатия пользователем кнопки **Back** (Назад) для возврата на ранее просмотренную страницу. Когда это происходит, браузер может извлечь часть или все содержимое этой страницы из своего локального кеша и сразу же снова отобразить страницу без необходимости доступа к удаленному веб-серверу и повторного получения той же информации.
- **Кеш-память инструкций и данных процессора.** Структуры кеш-памяти процессора рассматриваются в следующих разделах. Цель этой кеш-памяти — повысить скорость доступа к инструкциям и данным за счет устранения задержки, возникающей при доступе к модулям DRAM.

Кеш-память повышает производительность компьютера, поскольку многие алгоритмы, выполняемые операционными системами и приложениями, демонстрируют локальность ссылок. Термин "**локальность ссылок**" относится к повторному использованию данных, доступ к которым был получен недавно (**временнáя локальность**), а также к повторному использованию данных, находящихся в физической близости от данных, запрошенных ранее (**пространственная локальность**).

Если взять использование структуры TLB в качестве примера, то временная локальность применяется в нем путем сохранения данных трансляций виртуальных адресов в физические в течение некоторого времени после первоначального доступа к определенному страничному кадру. Любые дополнительные обращения к тому же страничному кадру в последующих инструкциях обеспечат намного более быстрый доступ к данным трансляции, пока эти данные в конечном счете не будут заменены в кеше данными другой трансляции.

Пространственную локальность TLB использует, ссылаясь на весь страничный кадр с помощью одной своей записи. Любые последующие обращения к разным адресам на той же странице выиграют по скорости от наличия записи TLB, полученной в результате первой ссылки на эту страницу.

Как правило, размер областей кеш-памяти невелик по сравнению с размером резервного хранилища. Устройства кеш-памяти рассчитаны на максимальную скорость, что обычно означает, что они являются более сложными и дорогостоящими в расчете на бит по сравнению с технологиями хранения данных, используемыми в резервном хранилище. Ввиду своего ограниченного размера устройства кеш-

памяти обычно быстро заполняются. Если в кеше нет доступного места для хранения новой записи, необходимо отбросить более раннюю запись. Для выбора записи кеша, которая будет перезаписана новой записью, контроллер кеша использует **политику замещения содержимого кеша**.

Цель кеш-памяти процессора — довести долю кеш-попаданий с течением времени до максимума, обеспечив тем самым максимально высокую среднюю скорость выполнения инструкций. Для достижения этой цели логика кеширования должна определять, какие инструкции и данные будут помещены в кеш и сохранены для будущего использования.

Логика кеширования процессора не гарантирует, что кешированный элемент данных когда-либо будет использоваться снова после того, как он был добавлен в кеш.

Эта логика основана на том, что благодаря временно́й и пространственной локальности существует достаточно высокая вероятность того, что обращение к кешированным данным последует в ближайшем будущем. На практике в современных процессорах кеш-попадания обычно происходят в 95–97% случаев обращений к памяти. Поскольку задержка кеш-памяти составляет лишь малую долю от задержки DRAM, высокая частота кеш-попаданий приводит к существенному повышению производительности по сравнению с архитектурой без кеш-памяти.

В следующих разделах обсуждаются технологии многоуровневого кеширования современных процессоров и некоторые политики замещения содержимого кеш-памяти, используемые при их реализации.

Многоуровневое кеширование в процессорах

За годы, прошедшие с момента появления персональных компьютеров, процессоры продемонстрировали значительное увеличение скорости обработки инструкций. Внутренние часы современных процессоров Intel и AMD почти в 1000 раз быстрее, чем у процессора 8088, использовавшегося в первом IBM PC. Для сравнения, скорость технологии памяти DRAM со временем увеличивалась гораздо медленнее. Исходя из этих тенденций, можно сказать, что если бы современный процессор имел прямой доступ к DRAM для всех своих инструкций и данных, он тратил бы существенную долю своего времени на ожидание ответа DRAM на каждый запрос.

Для того чтобы дать для этой темы некоторые приблизительные цифры, рассмотрим современный процессор, способный обрабатывать 32-битное значение данных из регистра процессора за 1 нс. Доступ к такому значению из DRAM может занять 100 нс. Несколько упростим ситуацию: если для каждой инструкции требуется одно обращение к ячейке памяти, а время выполнения каждой инструкции намного меньше времени доступа к памяти, можно ожидать, что цикл обработки, в котором выполняется доступ к требуемым данным из регистров процессора, будет работать в 100 раз быстрее, чем тот же алгоритм, обращающийся к основной памяти при выполнении каждой инструкции.

Теперь предположим, что в систему добавлена кеш-память со временем доступа 4 нс. Благодаря преимуществам кеш-памяти алгоритм, который обращается к

DRAM при выполнении каждой инструкции, будет испытывать снижение производительности, затрачивая на первое обращение к определенному адресу 100 нс, но последующие обращения к тем же и соседним адресам будут происходить со скоростью доступа к кеш-памяти — 4 нс. Доступ к кешу в четыре раза медленнее, чем доступ к регистрам, однако он все же в 25 раз быстрее доступа к DRAM. Этот пример показывает степень ускорения выполнения, достижимую за счет эффективного применения кеш-памяти в современных процессорах.

Высокопроизводительные процессоры обычно используют несколько уровней кеширования с целью достижения максимальной скорости выполнения инструкций. Аппаратные средства кеша процессора ограничены по размеру и производительности экономикой полупроводниковых технологий. Выбор оптимального сочетания типов и размеров кеш-памяти процессора при условии получения приемлемой для конечных пользователей цены является ключевой целью разработчиков процессоров.

В качестве основной памяти и для внутреннего хранения данных процессора обычно используются схемы оперативной памяти двух типов — динамическая (DRAM) и статическая (SRAM) оперативная память. Схемы DRAM имеют невысокую цену, но предлагают сравнительно медленный доступ, в основном из-за затрат времени на зарядку и разрядку конденсаторов битовых ячеек во время операций чтения и записи.

SRAM намного быстрее DRAM, но и намного дороже ее, чем обусловлено использование небольших объемов статической памяти в системах, где производительность имеет решающее значение. Конструкции схем DRAM оптимизированы по плотности, результатом чего является хранение максимально возможного количества битов на одной интегральной схеме DRAM. Конструкции схем SRAM оптимизированы по скорости, чтобы свести к минимуму время чтения или записи ячейки памяти. Кеш-память процессора обычно реализуется с использованием статической оперативной памяти.

Статическая оперативная память

Статическая оперативная память (static RAM, SRAM) предлагает значительно более быстрый доступ к данным по сравнению с DRAM, хотя и за счет значительно более сложной схемы. Битовые ячейки SRAM занимают гораздо больше места на кристалле интегральной схемы, чем ячейки DRAM, способные хранить эквивалентный объем данных. Как вы помните из *главы 4*, одна битовая ячейка DRAM состоит всего из одного полевого МОП-транзистора и одного конденсатора.

Стандартная схема для одного бита SRAM содержит шесть полевых МОП-транзисторов. Четыре из этих транзисторов используются для формирования двух элементов НЕ. Эти вентили основаны на КМОП-схеме, показанной на рис. 4.3 в *разд "Знакомство с полевыми МОП-транзисторами" главы 4*. Эти вентили обозначены как G_1 и G_2 на рис. 8.1.

Выход каждого элемента НЕ соединен со входом другого, образуя триггер. Большую часть времени на шине слов сохраняется низкий уровень, что поддерживает

транзисторные переключатели T_1 и T_2 в выключенном состоянии, изолируя эту пару вентилей. Пока на шине слов действует низкий уровень (и подача питания продолжается), вентили будут находиться в одном из двух состояний:

- **сохраненный бит равен 0**: на входе G_1 — низкий уровень, а на его выходе — высокий;
- **сохраненный бит равен 1**: на входе G_1 — высокий уровень, а на его выходе — низкий.

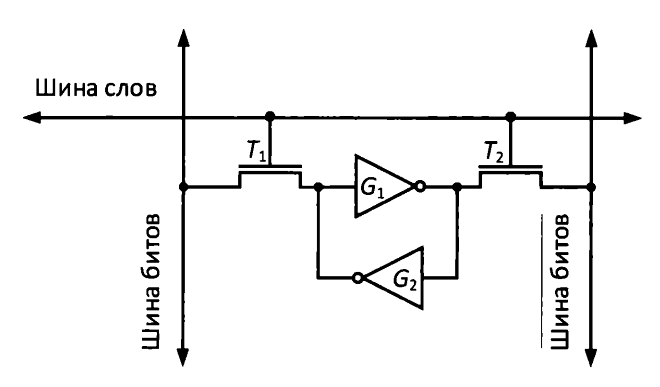


Рис. 8.1. *Схема ячейки SRAM*

Входные транзисторы (T_1 и T_2) функционируют как переключатели, которые соединяют шину битов с ячейкой для чтения и записи. Как и в случае с DRAM, перевод шины слов на высокий уровень обеспечивает доступ к битовой ячейке за счет уменьшения сопротивления на каждом входном транзисторе до очень малого значения. Для того чтобы считать содержимое ячейки, схема считывания измеряет напряжение между парой шин битов, обозначенных *Шина битов* и $\overline{\textit{Шина битов}}$ (где линия сверху означает инверсию — операцию НЕ). Сигналы этих двух шин всегда имеют противоположные значения, образуя дифференциальную пару. Измерение знака разности напряжений между двумя сигналами позволяет определить содержимое ячейки: 0 или 1.

При записи в битовую ячейку шина слов переводится на высокий уровень, а на шины битов подаются сигналы с противоположными уровнями напряжения, представляющие желаемое значение (0 или 1) для записи. Транзисторы, записывающие данные в шины битов, должны обладать существенно большей мощностью, чем транзисторы вентилей НЕ битовой ячейки. Благодаря этому желаемое значение можно записать в ячейку, даже если при этом необходимо подавить состояние триггера, чтобы переключить его в записываемое состояние.

Банк памяти SRAM представляет собой прямоугольную сетку строк и столбцов, как и в DRAM. Шина слов обеспечивает доступ ко всем битовым ячейкам SRAM в одной строке. Шины битов соединяются со всеми столбцами сетки битовых ячеек.

В отличие от DRAM, в SRAM не требуется периодическое обновление для сохранения хранящихся в ячейках данных. Именно по этой причине этот вариант памяти называют *статической* оперативной памятью.

В следующем разделе мы увидим, как с помощью SRAM создается первый уровень кеш-памяти процессора.

Кеш первого уровня

В кеш-памяти с многоуровневой архитектурой уровни нумеруются, начиная с 1. Кеш первого уровня (который также называют **кеш L1**) — это первая область кеш-памяти, к которой процессор обращается при запросе информации или элемента данных из памяти. Так как это первое обращение к кеш-памяти, кеш L1 обычно создается с использованием самой быстрой доступной технологии SRAM и физически расположен максимально близко к логической схеме процессора.

Акцент на скорости делает кеш L1 дорогостоящим и энергозатратным. Это означает, что он должен быть довольно небольшим по сравнению с основной памятью, особенно в решениях, где затраты являются важным фактором. Даже при скромных размерах быстрый кеш первого уровня может обеспечить существенное повышение производительности по сравнению с аналогичным процессором, в котором не используется кеширование.

Процессор (или блок управления памятью, MMU, если он присутствует) перемещает данные между DRAM и кешем в виде блоков данных фиксированного размера, называемых **строками кеша**. В компьютерах, оснащенных модулями DDR DRAM, размер строки кеша обычно равен 64 байтам. Такой же размер строки кеша обычно используется на всех уровнях кеш-памяти.

В современных процессорах кеш L1 часто разделяют на две секции одинакового размера: одну для инструкций и одну для данных. Такую конфигурацию называют **разделенной кеш-памятью**. В ней кеш инструкций первого уровня называют **I-кеш L1**, а кеш данных первого уровня — **D-кеш L1**. Процессор использует отдельные шины для доступа к каждому из этих кешей, тем самым реализуя важный аспект гарвардской архитектуры. Такая организация ускоряет выполнение инструкций за счет параллельного доступа к инструкциям и данным с учетом того, что кеш L1 используется для обоих видов доступа.

В современных процессорах применяют множество стратегий для организации кеш-памяти и управления ее работой. Простейшей конфигурацией кеш-памяти является кеш с прямым отображением, представленный в следующем разделе.

Кеш с прямым отображением

Кеш с прямым отображением — это блок памяти, организованный в виде одномерного массива наборов кеша, где каждый адрес в основной памяти сопоставлен с одним набором в кеше. Каждый **набор кеша** состоит из следующих элементов:

- строка кеша, содержащая блок данных, считанных из основной памяти;
- значение тега, указывающее местоположение в основной памяти, соответствующее кешированным данным;
- бит допустимости, указывающий, содержит ли этот набор кеша данные.

В этом примере представлен кеш инструкций, доступный только для чтения. Кеши данных для чтения и записи мы рассмотрим в следующем разделе.

Бывают случаи, когда кеш не содержит данных, например сразу после включения процессора. Бит допустимости каждого набора кеша изначально имеет нулевое значение, что указывает на отсутствие данных в наборе. Когда бит допустимости равен 0, использование этого набора для поиска запрещено. При загрузке данных в набор кеша, аппаратные средства устанавливают бит допустимости.

В качестве примера мы будем использовать небольшой I-кеш L1 размером 512 байт. Поскольку это кеш инструкций, доступный только для чтения, для него не требуется поддержка возможности записи в память. Размер строки кеша составляет 64 байта. Деление 512 байт на 64 байта для каждого набора дает 8 наборов кеша.

64 байта в каждом наборе — это 2^6 байт, т. е. 6 наименее значимых битов адреса определяют местоположение в строке кеша. Для выбора одного из восьми наборов кеша требуются три дополнительных бита адреса.

Исходя из этой информации, на рис. 8.2 показано разделение 32-разрядного адреса физической памяти на компоненты: тег, номер набора и байтовое смещение строки кеша.

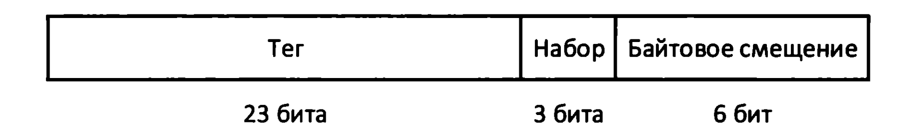


Рис. 8.2. *Компоненты 32-разрядного адреса физической памяти*

Каждый раз, когда процессор считывает инструкцию из DRAM (что необходимо всякий раз, когда инструкция отсутствует в кеше), MMU считывает 64-байтовый блок, содержащий адресованную ячейку, и сохраняет его в наборе I-кеша L1, выбранном тремя битами компонента "Набор", показанного на рис. 8.2. Старшие 23 бита адреса хранятся в поле "Тег" набора кеша, а бит **допустимости** установлен.

По мере того, как процессор извлекает каждую последующую инструкцию, устройство управления использует три бита компонента "Набор" в адресе инструкции, чтобы выбрать набор кеша для сравнения. Аппаратные средства сравнивают старшие 23 бита адреса выполняемой инструкции со значением тега, хранящимся в выбранном наборе кеша. Если эти значения совпадают, то произошло кеш-попадание, и процессор считывает инструкцию из строки кеша. Если происходит кеш-промах, MMU считывает строку из DRAM в соответствующий набор кеша (перезаписывая любые существующие данные для этого набора) и передает инструкцию устройству управления для выполнения.

На рис. 8.3 представлены организация всего 512-байтового кеша и его связь с тремя полями в 32-разрядном адресе инструкции.

Для того чтобы продемонстрировать, почему кеш с прямым отображением может обеспечить высокую частоту попаданий, предположим, что мы запускаем программу, содержащую цикл, начинающийся с адреса физической памяти 8000h (здесь для

простоты мы игнорируем старшие 16 бит 32-разрядного адреса) и содержащий 256 байт кода. В этом цикле инструкции выполняются последовательно от начала до конца 256-байтового диапазона, после чего осуществляется возврат к началу цикла для следующей итерации.

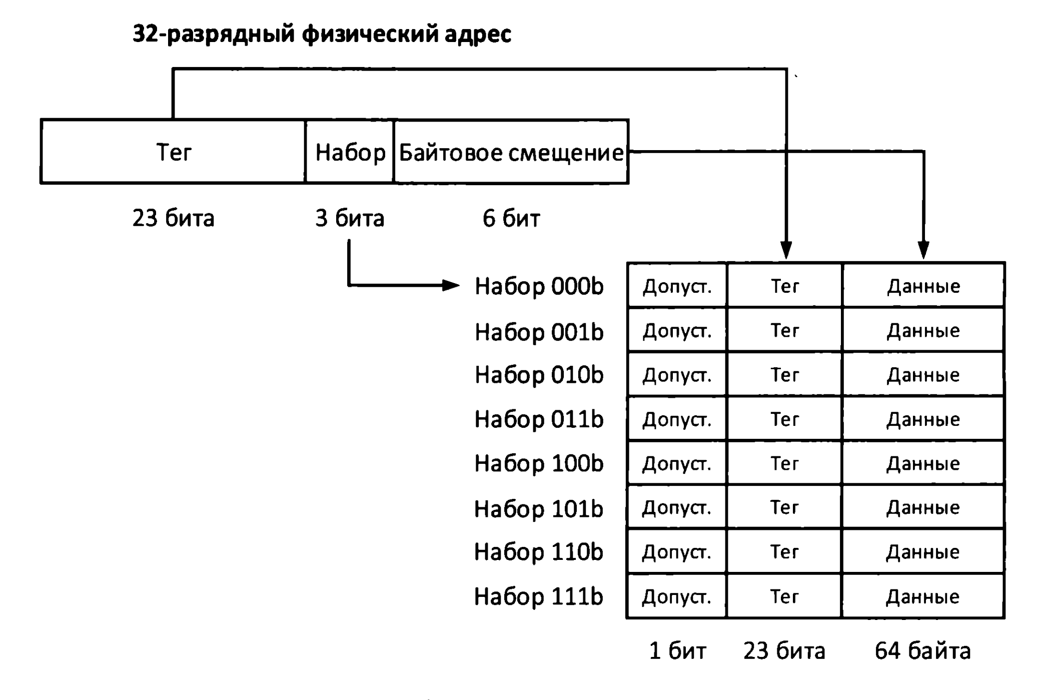


Рис. 8.3. *Связь 32-разрядного физического адреса с кеш-памятью*

Адрес 8000h содержит 000b в поле "Набор", поэтому этот адрес сопоставляется с первым набором кеша, как показано на рис. 8.3. При первом проходе через цикл MMU извлекает из DRAM 64-байтовую строку кеша и сохраняет ее в первом наборе кеша (со значением поля "Набор", равным 000b). По мере выполнения остальных инструкций, хранящихся в том же 64-байтовом блоке, каждая из них будет извлекаться непосредственно из кеша.

При переходе процесса выполнения ко второму 64-байтовому блоку потребуется еще одно обращение к DRAM для считывания данных. К моменту достижения конца цикла наборы с 000b по 011b оказываются заполненными 256 байтами кода для данного цикла. Для оставшихся проходов через этот цикл, с учетом предположения, что поток выполняется без прерываний, процессор достигнет 100% кеш-попаданий и максимальной скорости выполнения инструкций.

С другой стороны, если инструкции в цикле потребляют значительно больше памяти, то выгода от кеширования уменьшится. Предположим, что инструкции цикла занимают 1024 байта, т. е. вдвое больше размера кеша. В цикле, с его начала и до конца, выполняется одна и та же последовательность. Когда адреса инструкций достигают середины цикла, кеш оказывается заполненным первыми 512 байтами инструкций. В начале следующей строки кеша после средней точки адрес будет

равен 8000h плюс 512, что равно 8200h. Группа битов "Набор" в адресе 8200h — та же, что и в адресе 8000h, что приводит к перезаписи строки кеша для адреса 8000h строкой кеша для адреса 8200h. Каждая последующая строка кеша будет перезаписываться по мере выполнения кода второй половины цикла.

Несмотря на то что все области кешированной памяти перезаписываются при каждом прохождении цикла, структура кеш-памяти по-прежнему обеспечивает значительную выгоду, т. к. после считывания из DRAM каждая 64-байтовая строка остается в кеше и доступна для использования по мере выполнения инструкций. Недостатком в этом примере является повышенная частота промахов кеша. Это влечет за собой существенное снижение производительности, поскольку, как мы видели, доступ к инструкции в DRAM может быть в 25 (или более) раз медленнее, чем доступ к той же инструкции в I-кеше L1.

ТЕГИ ВИРТУАЛЬНЫХ И ФИЗИЧЕСКИХ АДРЕСОВ ПРИ КЕШИРОВАНИИ

Пример в этом разделе предполагает, что кеш-память использует адреса физической памяти для пометки записей кеша. Это означает, что в системах с поддержкой виртуальной памяти со страничной организацией адреса, используемые при поиске в кеше, являются результатом трансляции виртуальных адресов в физические. Разработчик процессора должен выбрать, какие адреса следует использовать для целей кеширования — виртуальные или физические.



В кеш-памяти современных процессоров на одном или нескольких уровнях кеширования нередко используются теги виртуальных адресов, а на остальных уровнях — теги физических адресов. Одним из преимуществ использования тегов виртуальных адресов является скорость, поскольку при обращении к кешу не требуется трансляция виртуального адреса в физический. Как мы видели, при трансляции виртуального адреса в физический выполняется поиск в буфере TLB и, возможно, поиск по таблице страниц в случае промаха TLB. Однако использование тегов виртуальных адресов создает другие проблемы, такие как **множественность ссылок**, которая возникает, когда один и тот же виртуальный адрес ссылается на разные физические адреса.

Как и во многих других аспектах оптимизации производительности процессора, это компромисс, который следует учитывать при проектировании системы кеширования.

Этот пример был упрощен за счет предположения, что выполнение инструкций происходит линейно с начала и до конца цикла без каких-либо обходных путей. В реальном коде часто происходят вызовы функций в различных областях памяти приложения и обращения к системным библиотекам.

В дополнение к этим факторам информация в кеше часто перезаписывается при выполнении других системных действий, таких как обработка прерываний и переключение контекста потока, что приводит к более высокой частоте промахов кеша. Это влияет на производительность приложения, поскольку код основной ветви должен выполнять дополнительные обращения к DRAM для перезагрузки кеша после каждого отклонения, которое приводило к изменениям записей в кеше.

Одним из способов уменьшить последствия отклонений от прямолинейного выполнения кода является настройка нескольких кешей, работающих параллельно. Такую конфигурацию, которая обсуждается в следующем разделе, называют **наборно-ассоциативным кешем**.

Наборно-ассоциативный кеш

В двухканальном наборно-ассоциативном кеше память делится на два кеша одинакового размера. Каждый из них содержит половину общего количества записей кеша с прямым отображением того же общего размера. При каждом доступе к памяти аппаратные средства обращаются к обоим кешам параллельно, и попадание может произойти в любом из них. Следующая схема иллюстрирует одновременное сравнение тега 32-разрядного адреса с тегами, содержащимися в двух I-кешах L1.

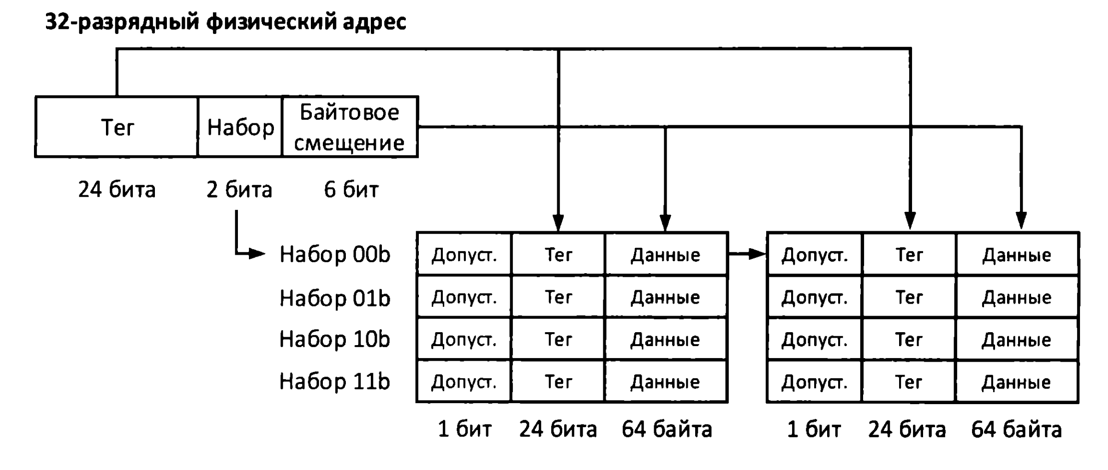


Рис. 8.4. *Работа наборно-ассоциативного кеша*

Конфигурация кеша, показанная здесь, имеет тот же размер строки кеша (64 байта), что и кеш на рис. 8.3, но вдвое меньше наборов на кеш. Общий размер кеш-памяти такой же, как и в предыдущем примере: 64 байта на строку, умноженные на 4 строки, умноженные на 2 кеша, равны 512 байтам. Поскольку теперь существует четыре набора, поле "Набор" в физическом адресе уменьшается до 2 бит, а поле "Тег" увеличивается до 24 бит. Каждый набор состоит из двух строк кеша, по одной в каждом из двух кешей.

При кеш-промахе логика управления памятью должна выбрать, какую из двух таблиц кеша использовать в качестве места назначения для данных, загружаемых из

DRAM. Общепринятый метод заключается в том, чтобы отслеживать, к какому из соответствующих наборов в двух таблицах дольше не было обращений, и перезаписать эту запись. Эта политика замещения, называемая **LRU** (least recently used — замена наиболее давно использовавшегося элемента), требует аппаратной поддержки для отслеживания того, какая строка кеша не была востребована дольше всего. Политика LRU основана на эвристике временной локальности, которая гласит, что данные, к которым не обращались в течение длительного времени, с меньшей вероятностью будут затребованы снова в ближайшее время.

Другой способ выбора между двумя таблицами — просто чередовать их при последовательных вставках в кеш.

Аппаратные средства для реализации этой политики замены проще, чем для политики LRU, но такой подход может повлиять на производительность из-за того, что произвольный выбор строки может быть неверным. Выбор политики замещения содержимого кеша представляет собой еще одну область компромисса между повышенной сложностью аппаратных средств и постепенным повышением производительности.

В двухканальном наборно-ассоциативном кеше две строки кеша из разных физических местоположений с совпадающими полями "Набор" могут присутствовать в кеше одновременно, при условии, что обе строки кеша допустимы. В то же время, по сравнению с кешем с прямым отображением того же общего размера, каждый из двухканальных наборно-ассоциативных кешей имеет вдвое меньший размер. Это еще один компромисс проектирования: количество уникальных значений поля "Набор", которые могут храниться в двухканальном наборно-ассоциативном кеше, меньше по сравнению с прямым отображением, но при этом возможно одновременное кеширование нескольких строк кеша с одинаковыми полями "Набор".

Конфигурация кеша, обеспечивающая наилучшую общую производительность системы, будет зависеть от особенностей доступа к памяти, связанных с типами операций обработки, выполняемых в системе.

Наборно-ассоциативный кеш может содержать более двух кешей, рассмотренных в этом примере. Современные процессоры часто имеют 4, 8 или 16 параллельных кешей, которые называют 4-канальными, 8-канальными и 16-канальными наборно-ассоциативными кешами. Эти кеши называются **наборно-ассоциативными**, т. к. поле "Тег" адреса одновременно относится ко всем строкам кеша в наборе. Кеш с прямым отображением реализует одноканальный наборно-ассоциативный кеш.

Преимущество многоканальных наборно-ассоциативных кешей по сравнению с кешами с прямым отображением заключается в том, что они, как правило, имеют более высокую частоту кеш-попаданий, что в большинстве практических случаев ведет к повышению производительности системы по сравнению с кешами с прямым отображением. Если многоканальные наборно-ассоциативные кеши обеспечивают лучшую производительность, чем одноканальные кеши с прямым отображением, то почему бы тогда не повысить уровень ассоциативности? Доведенная до предела, эта прогрессия заканчивается полностью ассоциативным кешированием.

Полностью ассоциативный кеш

Предполагая, что количество строк в кеше равно степени двойки, повторяющееся разделение общей кеш-памяти на большее количество меньших параллельных кешей, пока каждый кеш не будет содержать только одну строку, приводит к получению **полностью ассоциативного кеша**.

В нашем примере 512-байтового кеша с 64 байтами на строку кеша этот процесс приведет к созданию восьми параллельных кешей, каждый из которых содержит только один набор.

В этой архитектуре каждое обращение к памяти приводит к параллельному сравнению со значениями тегов, хранящимися во всех строках кеша. Полностью ассоциативный кеш с использованием эффективной политики замены, такой как LRU, может обеспечить очень высокую частоту кеш-попаданий, хотя и при значительных затратах на сложность схемы и соответствующее потребление площади кристалла.

В решениях, чувствительных к энергопотреблению, таких как мобильные устройства с питанием от аккумулятора, повышение сложности схемы полностью ассоциативного кеша приводит к увеличению разряда аккумулятора. В настольных компьютерах и облачных серверах энергопотребление процессора должно быть сведено к минимуму, чтобы избежать необходимости в чрезвычайных мерах для охлаждения и свести к минимуму счета за электроэнергию для поставщиков облачных услуг, эксплуатирующих тысячи серверов. Ввиду таких затрат полностью ассоциативный кеш редко используется в качестве кеша инструкций и данных в современных процессорах.

Концепция, лежащая в основе полностью ассоциативного кеша, может показаться знакомой, поскольку это та же концепция, которая применяется в буфере TLB, описанном в *главе 7*. TLB обычно представляет собой полностью ассоциативный кеш, содержащий результаты трансляции виртуальных адресов в физические. Использование полностью ассоциативного кеширования в TLB имеет свои недостатки, описанные в предыдущем абзаце: сложность схемы, потребность в дополнительной площади кристалла и повышенное энергопотребление, — однако преимущество в виде повышения производительности, которое обеспечивает TLB, настолько существенно, что полная ассоциативность применяется практически во всех высокопроизводительных процессорах, где реализована виртуальная память со страничной организацией.

Наше обсуждение до этого момента было сосредоточено на кешировании инструкций, которое обычно является процессом с доступом только для чтения. Функциональные требования к кешу данных во многом аналогичны требованиям к кешу инструкций с одним важным дополнением: помимо чтения из памяти данных процессор должен быть свободен для записи в память данных. Это тема следующего раздела.

Политики записи в кеш процессора

В процессорах с разделенной кеш-памятью D-кеш L1 аналогичен по структуре I-кешу L1 за исключением того, что схема должна позволять процессору не только считывать данные из памяти, но и записывать их в память. Каждый раз, когда процессор записывает значение данных в память, он должен обновить строку D-кеша L1, содержащую этот элемент данных, и в какой-то момент времени он также должен обновить ячейку DRAM в физической памяти, содержащую эту строку кеша. Запись в память DRAM, как и чтение из нее, является медленным процессом по сравнению со скоростью записи в D-кеш L1.

Ниже перечислены наиболее распространенные политики записи в кеш в современных процессорах.

- **Сквозная запись (write-through).** Эта политика предусматривает немедленное обновление DRAM каждый раз, когда процессор записывает данные в память.
- **Отложенная запись (write-back).** При использовании этой политики измененные данные удерживаются в кеше до тех пор, пока соответствующая строка не будет вытеснена из кеша. Для каждой строки кеша с отложенной записью должен быть предусмотрен дополнительный бит состояния, указывающий, были ли изменены данные с момента их считывания из DRAM. Этот бит называется **битом факта записи**. Установленное состояние этого бита указывает, что данные в этой строке были изменены, и система должна записать их в DRAM, прежде чем соответствующая строка кеша может быть освобождена для повторного использования.

Политика отложенной записи обычно приводит к повышению производительности системы, поскольку дает процессору возможность выполнять несколько операций записи в одну и ту же строку кеша без необходимости обращаться к основной памяти при каждой записи. В системах с несколькими ядрами или несколькими процессорами использование кеширования с отложенной записью создает сложности, поскольку D-кеш L1, принадлежащий ядру, которое выполняет запись в ячейку памяти, будет содержать данные, отличающиеся от содержимого соответствующей ячейки DRAM и от любых копий в кешах других процессоров или ядер. Такое несоответствие содержимого памяти будет сохраняться до тех пор, пока эта строка кеша не будет записана в DRAM и обновлена в любом из кешей процессора, где она находится.

Как правило, недопустимо, чтобы разные ядра процессора извлекали различные значения при обработке одной и той же ячейки памяти, поэтому необходимо предусмотреть решение, гарантирующее, что все процессоры всегда видят одни и те же данные в каждой ячейке памяти. Задача поддержания идентичных представлений содержимого памяти на нескольких процессорах называется задачей **согласования кеша**.

В многоядерном процессоре можно было бы решить эту проблему путем организации общего доступа всех ядер к одному и тому же D-кешу L1, но современные

многоядерные процессоры обычно реализуют отдельный D-кеш L1 для каждого ядра, чтобы довести скорость доступа до максимума. В многопроцессорных системах с общей основной памятью, где процессоры находятся на отдельных интегральных схемах, и в многоядерных процессорах, которые не используют общий кеш L1, эта задача отличается большей сложностью.

В некоторых многопроцессорных архитектурах для поддержания согласованности кеша применяется **отслеживание**. Наблюдая за памятью, процессор отслеживает операции записи в память, выполняемые другими процессорами. Когда происходит запись в ячейку памяти, присутствующую в кеше отслеживающего процессора, отслеживающее устройство выполняет одно из двух действий: оно может сделать недействительной свою строку кеша, установив бит **допустимости** в состояние `false`, или занести в эту строку данные, записанные другим процессором. Если строка кеша признана недопустимой, следующее обращение к диапазону адресов этой строки вызовет доступ к DRAM для получения данных, измененных другим процессором.

Отслеживание может быть эффективным в системах с ограниченным числом процессоров, но его сложно масштабировать для систем, содержащих десятки или сотни процессоров. Это связано с тем, что каждый процессор должен постоянно отслеживать операции записи всех других процессоров. В системах с большим количеством процессоров должны быть реализованы другие, более сложные протоколы согласования кеша.

Кеши процессора второго и третьего уровней

До настоящего момента наше обсуждение было сосредоточено на кешах инструкций и данных первого уровня — L1. Эти кеши спроектированы так, чтобы быть максимально быстрыми, но акцент на скорости ограничивает их размер из-за сложности схемы кеша и требований к питанию.

Из-за большой разницы в задержке кеша L1 и DRAM разумно задаться вопросом: может ли промежуточный уровень кеширования между L1 и DRAM повысить производительность по сравнению с процессором, содержащим только кеш L1? Ответ — да, добавление кеша L2 обеспечивает существенное повышение производительности.

На кристаллах современных высокопроизводительных процессоров обычно содержится банк кеша L2 значительного объема. В отличие от кеша L1, кеш L2 обычно объединяет инструкции и данные в одной области памяти, представляя архитектуру фон Неймана, а не характерное для гарвардской архитектуры разделение в структуре кеша L1.

Кеш L2 обычно работает медленнее, чем кеш L1, но все же намного быстрее, чем прямой доступ к DRAM. Кеш L2 использует битовые ячейки SRAM с той же базовой схемой, что показана на рис. 8.1, однако в схемном решении L2 уделяется особое внимание уменьшению занимаемой на кристалле площади в расчете на бит и снижению энергопотребления по сравнению со схемным решением L1. Эти моди-

фикации позволяют сделать кеш L2 намного больше по объему, чем кеш L1, но они также делают его значительно медленнее.

Типичный кеш L2 может быть в четыре или восемь раз больше, чем I-кеш и D-кеш L1 вместе взятые, при этом время доступа у него будет в 2–3 раза больше, чем у L1. В зависимости от конструкции процессора кеш L2 может включать или не включать в себя кеш L1. **Инклюзивный кеш L2**, помимо других строк, всегда содержит строки кеша, имеющиеся в кеше L1.

Как мы уже видели, каждый раз, когда процессор обращается к памяти, он сначала обращается к кешу L1. Если происходит промах кеша L1, следующим пунктом для проверки является кеш L2. Поскольку L2 больше, чем L1, существует значительная вероятность того, что данные будут найдены именно там. Если искомая строка кеша отсутствует в L2, требуется доступ к DRAM. Процессоры обычно выполняют некоторые из этих шагов параллельно, чтобы гарантировать, что промах кеша не приведет к длительному последовательному поиску запрошенных данных.

Каждый промах кеша L1 и L2 приводит к выполнению операции чтения DRAM, которая заполняет соответствующие строки кеша в L1 и L2 при условии, что L2 является инклюзивным. За каждой выполняемой процессором операцией записи в память данных рано или поздно должно последовать обновление как L1, так и L2. D-кеш L1 реализует политику записи в кеш (обычно это политика сквозной или отложенной записи), чтобы определить, когда должно быть выполнено обновление кеша L2. В кеше L2, аналогичным образом, реализована собственная политика записи, помогающая определить, когда следует записывать измененные строки кеша в DRAM.

Если использование двух уровней кеша помогает повысить производительность, стоит ли ограничиваться лишь ими? В действительности в большинстве современных высокопроизводительных процессоров реализованы три (или даже более!) уровня кеша на кристалле. Как и при переходе от L1 к L2, переход от L2 к L3 влечет за собой увеличение объема памяти и уменьшение скорости доступа. Как и L2, кеш L3 обычно объединяет инструкции и данные в одной области памяти. В процессорах потребительского уровня кеш L3 обычно представляет собой SRAM объемом в несколько мегабайт с временем доступа в 3–4 раза медленнее, чем кеш L2.

ВЗГЛЯД ПРОГРАММИСТА НА КЕШ-ПАМЯТЬ



Разработчикам программного обеспечения не нужно предпринимать какие-либо шаги, чтобы воспользоваться преимуществами кеш-памяти, однако понимание среды, в которой работает их программное обеспечение, может помочь повысить производительность кода, выполняемого процессорами с несколькими уровнями кеш-памяти. При наличии достаточной гибкости определение размеров структур данных и структурирование внутренних циклов кода для работы в пределах предполагаемых размеров кешей L1, L2 и L3 может привести к значительному увеличению скорости выполнения.

> **На производительность любого фрагмента кода влияет множество факторов, связанных с архитектурой процессора и поведением системного программного обеспечения, поэтому лучший способ определить оптимальный алгоритм среди нескольких жизнеспособных альтернатив — это тщательное сравнение производительности каждого из них.**

Многоуровневая иерархия кеш-памяти в современных процессорах приводит к значительному повышению производительности по сравнению с аналогичной системой без кеш-памяти. Кеширование позволяет самым быстрым процессорам большую часть времени работать с минимальной задержкой доступа к DRAM. Несмотря на то что кеш-память значительно усложняет конструкцию процессора и потребляет много ценного места на кристалле и энергии, производители процессоров решили, что использование многоуровневой архитектуры кеш-памяти вполне оправдывает эти затраты.

Кеш-память ускоряет выполнение инструкций за счет уменьшения задержки доступа к памяти при извлечении инструкций и данных, на которые ссылаются эти инструкции.

Следующая область повышения производительности, которую мы рассмотрим, — это возможности оптимизации в процессоре, помогающие увеличить скорость выполнения инструкций. Основным методом, который применяется в современных процессорах для достижения прироста производительности таким способом, является конвейеризация.

Конвейерная обработка инструкций

Прежде чем представить конвейерную обработку, давайте разобьем выполнение одной инструкции процессора на последовательность отдельных шагов.

- **Извлечение.** Устройство управления процессора получает доступ к адресу памяти, по которому находится следующая инструкция для выполнения, как определено предыдущей инструкцией, либо из предопределенного значения в результате сброса программного счетчика сразу после включения питания, либо в ответ на прерывание. Считав этот адрес, устройство управления загружает опкод инструкции во внутренний регистр инструкций процессора.
- **Декодирование.** По полученному опкоду устройство управления определяет действия, которые необходимо предпринять во время выполнения инструкции. Эти действия могут включать в себя обращение к АЛУ и потребовать доступа к регистрам или ячейкам памяти для чтения или записи.
- **Выполнение.** Устройство управления выполняет запрошенную операцию, вызывая при необходимости операцию АЛУ.
- **Запись результатов.** Устройство управления записывает результаты выполнения инструкции в регистры или ячейки памяти. После этого в программный счетчик загружается адрес следующей инструкции, которая должна быть выполнена.

Процессор выполняет повторяющийся цикл извлечения, декодирования, выполнения и записи с момента подачи питания до выключения компьютера. В относительно простом процессоре, таком как 6502, каждый из этих этапов выполняется процессором как фактически изолированные последовательные операции.

С точки зрения программиста, каждая инструкция проходит все эти этапы перед началом выполнения следующей инструкции. После завершения записи результаты и побочные эффекты каждой инструкции доступны в регистрах, памяти и флагах процессора для немедленного использования следующей инструкцией. Это простая модель выполнения, и работающие программы могут с уверенностью считать, что когда начинается выполнение инструкции, все действия предыдущих инструкций завершены.

Схема на рис. 8.5 является примером выполнения процессором инструкции, для каждого из этапов обработки которой (извлечение — декодирование — выполнение — запись) требуется один тактовый цикл. Обратите внимание, что каждый шаг на этой диаграмме обозначен первой буквой названия этапа: И, Д, В или З.

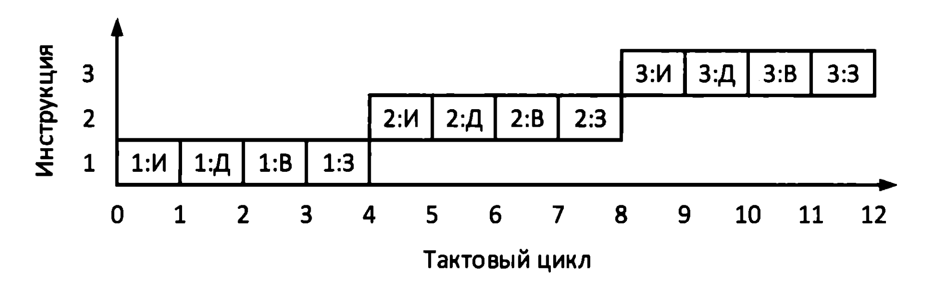


Рис. 8.5. *Последовательное выполнение инструкций*

Для выполнения каждой инструкции требуются четыре такта. На аппаратном уровне процессор, представленный на рис. 8.5, состоит из четырех подсистем выполнения, каждая из которых становится активной во время одного из четырех тактовых циклов инструкции. Логика обработки, связанная с каждым из этих этапов, предусматривает считывание входной информации из памяти и из регистров процессора и сохранение промежуточных результатов в защелках для использования на последующих этапах выполнения. После завершения выполнения каждой инструкции следующая инструкция начинает выполняться в следующем тактовом цикле.

Количество выполненных **инструкций за такт процессора** (instructions per clock, IPC) представляет собой показатель производительности, характеризующий, насколько быстро процессор выполняет инструкции относительно своей тактовой частоты. Процессору в примере на рис. 8.5 требуется четыре такта на одну инструкцию, т. е. показатель IPC для него равен 0,25.

Возможность повышения производительности в этом примере возникает из-за того, что схемы, задействованные в каждом из четырех этапов, простаивают, пока выполняются остальные три этапа. Предположим, что вместо этого, как только схема извлечения заканчивает извлечение одной инструкции, она немедленно начинает извлечение следующей инструкции. На рис. 8.6 показан процесс, когда аппаратные

средства, участвующие в каждом из четырех этапов выполнения инструкции, переходят от обработки одной инструкции к следующей на каждом тактовом цикле.

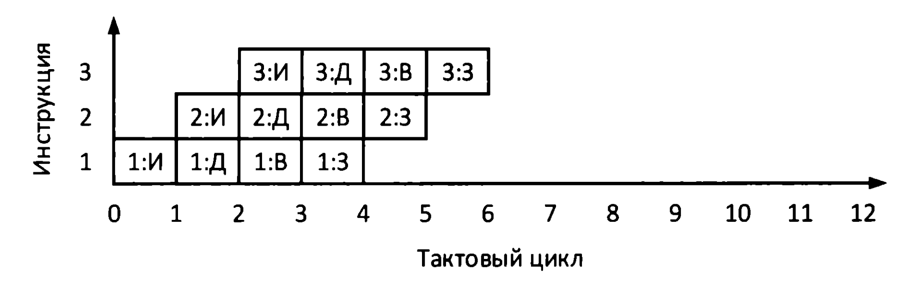


***Рис. 8.6.** Конвейерное выполнение инструкций*

Эта процедура выполнения называется **конвейерной**, т. к. инструкции поступают и проходят этапы выполнения от начала до конца, подобно изделиям, перемещающимся по реальному конвейеру. Конвейер процессора содержит множество инструкций, одновременно находящихся на различных этапах выполнения. Причина, побудившая пойти на это, очевидна из предыдущего примера: процессор теперь выполняет одну инструкцию за тактовый цикл, а IPC составляет 1,0, что дает четырехкратное увеличение скорости по сравнению с моделью выполнения без конвейерной обработки, представленной на рис. 8.5. Аналогичные уровни повышения производительности достигаются при обработке реальных данных с помощью методов конвейеризации.

Помимо наложения выполнения последовательных инструкций посредством конвейеризации, могут существовать и другие возможности продуктивного использования подсистем процессора, которые в противном случае могут простаивать. Инструкции процессора делятся на несколько различных категорий, каждая из которых требует задействования разных частей схемы процессора. Вот некоторые примеры.

- **Инструкции ветвления.** Инструкции условного и безусловного ветвления управляют программным счетчиком, чтобы установить адрес следующей инструкции, подлежащей выполнению.
- **Инструкции целочисленных вычислений.** Эти инструкции, реализующие целочисленную арифметику и побитовые операции, обращаются к целочисленной части АЛУ.
- **Инструкции вычислений с плавающей запятой.** Операции с плавающей запятой в процессорах, которые обеспечивают аппаратную поддержку этих операций, обычно используют схему, отделенную от целочисленной части АЛУ.

За счет повышения сложности логики планирования инструкций процессора может оказаться возможным инициировать выполнение двух инструкций за один и тот же тактовый цикл, если они используют независимые ресурсы обработки. Например,

обычной практикой является отправление процессорами инструкций вычислений с плавающей запятой в **математический сопроцессор** (floating point unit, FPU) для выполнения параллельно с инструкциями без плавающей запятой, которые обрабатываются ядром основного процессора.

На практике многие современные процессоры содержат несколько копий подсистем, таких как целочисленные АЛУ, для поддержки одновременного выполнения нескольких инструкций. В этой архитектурной парадигме, называемой **обработкой с множественной выдачей инструкций** (multiple-issue processing), процессор инициирует выполнение нескольких инструкций одновременно. На рис. 8.7 показан пример множественной выдачи, при которой на каждом такте процессора инициируется выполнение двух инструкций.

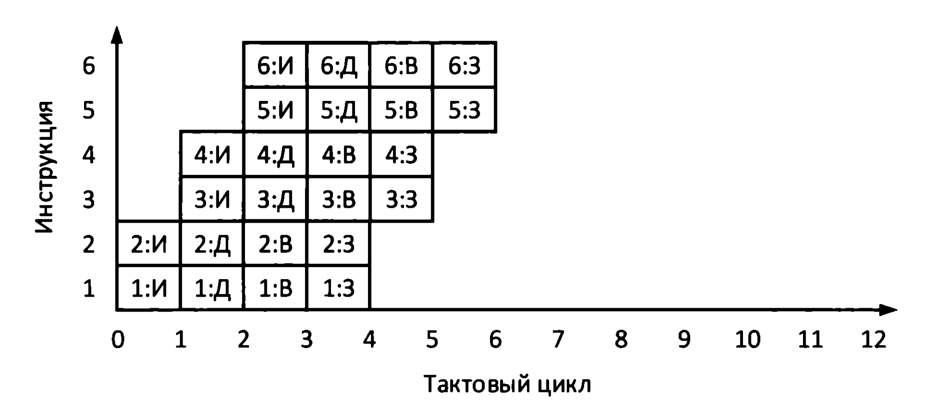


Рис. 8.7. Конвейерное выполнение с множественной выдачей инструкций

Эта модель выполнения удваивает количество инструкций, выполняемых за такт на одноканальном конвейере рис. 8.6, что позволяет довести IPC до 2,0. Данный подход представляет **суперскалярный процессор**, который может выдавать (другими словами, начинать выполнение) более одной инструкции за такт. **Скалярный процессор**, для сравнения, может выдавать только одну инструкцию за такт. Для того чтобы прояснить ситуацию: на рис. 8.5 и 8.6 представлено поведение скалярного процессора, а на рис. 8.7 — суперскалярная обработка. Суперскалярный процессор реализует **параллелизм на уровне инструкций** (instruction-level parallelism, ILP) для увеличения скорости выполнения. Практически все современные высокопроизводительные процессоры общего назначения являются суперскалярными.

Суперконвейеризация

На примере рассмотренной ранее скалярной, не конвейерной модели обработки, представленной на рис. 8.5, мы можем обсудить вопрос выбора тактовой частоты процессора. При отсутствии опасений по поводу энергопотребления и рассеивания тепла желательно, как правило, использовать максимально возможную тактовую частоту процессора. Верхний предел тактовой частоты для процессора на рис. 8.5

определяется самым длительным по времени путем через каждую из четырех подсистем, участвующих в этапах выполнения инструкций. Различные инструкции могут иметь совершенно разные требования к времени выполнения. Например, для выполнения инструкции снятия флага процессора требуется совсем небольшое время, в то время как для получения выходных данных инструкции 32-разрядного деления может потребоваться гораздо больше времени.

Неэффективно ограничивать общую скорость выполнения инструкций системы на основе наихудшего времени выполнения одной инструкции. Вместо этого разработчики процессоров изыскивают возможности разбить выполнение сложных инструкций на некоторое количество последовательных шагов. Этот подход называется **суперконвейеризацией**. Суперконвейеризация состоит в увеличении количества этапов конвейера за счет разбиения сложных этапов на множество более простых. Суперконвейеризация — это, по сути, конвейер процессора со многими этапами, возможно исчисляющихся десятками. В дополнение к тому, что современные высокопроизводительные процессоры являются суперскалярными, они, как правило, поддерживают и суперконвейеризацию.

Разбиение конвейера на большее количество этапов суперконвейера позволяет упростить каждый этап, сокращая время, необходимое для выполнения каждого этапа. Благодаря быстрее выполняемым этапам можно увеличить тактовую частоту процессора. Суперконвейеризация обеспечивает увеличение скорости выполнения инструкций в соответствии с процентным увеличением тактовой частоты процессора.

Наборы инструкций процессоров для **компьютеров с сокращенным набором инструкций** (reduced instruction set computer, RISC) разработаны для поддержки эффективной конвейеризации. Большинство RISC-инструкций выполняют простые операции, такие как перемещение данных между регистрами и памятью или сложение содержимого двух регистров. RISC-процессоры обычно имеют более короткие конвейеры по сравнению с процессорами для **компьютеров с полным набором инструкций** (complex instruction set computer, CISC). CISC-процессоры и их более обширные и сложные наборы инструкций извлекают выгоду из более длинных конвейеров, разбивая длительно выполняющиеся инструкции на серию последовательных этапов.

По большей части сложность реализации эффективной конвейерной обработки в процессорах, использующих устаревшие наборы инструкций, таких как *x*86, заключается в том, что первоначальный набор инструкций не полностью учитывал потенциал более поздних усовершенствований, таких как суперскалярная обработка и суперконвейеризация. В результате в современных *x*86-совместимых процессорах значительная часть площади кристалла выделена для сложной логики, реализующей эти функции повышения производительности для инструкций, которые не были предназначены для работы в такой среде.

Если разбиение конвейера на один или два десятка этапов приводит к существенному повышению производительности, почему бы не продолжить это разбиение на сотни или даже тысячи более мелких этапов, чтобы достичь еще большей производительности? Ответ: наличие конфликтов конвейеризации.

Конфликты конвейеризации

Реализация конвейерной обработки в процессоре общего назначения не так проста, как может показаться из предыдущего обсуждения этой темы. Если результат инструкции зависит от результата предыдущей инструкции, то текущей инструкции приходится ждать, пока не станет доступен результат предыдущей инструкции. Рассмотрим следующий код *x*86:

```
inc eax
add ebx, eax
```

Предположим, что эти две инструкции выполняются на процессоре, работающем как показано на рис. 8.6 (одноканальная конвейеризация). Первая инструкция увеличивает содержимое регистра eax на единицу, а вторая — добавляет полученное значение к содержимому регистра ebx. Инструкция add не может выполнить операцию сложения до тех пор, пока не будет доступен результат операции увеличения на 1 в предыдущей инструкции. Если этап выполнения второй инструкции (обозначенный как **2:В** на рис. 8.6) не может быть выполнен до тех пор, пока не завершится этап записи результата первой инструкции (**1:З** на рис. 8.6), у этапа 2:В нет другого выбора, кроме как дождаться завершения этапа 1:З. Ситуации, подобные этой, когда конвейер не может обработать следующий этап выполнения инструкции из-за отсутствия необходимой информации, называют **конфликтами конвейеризации**.

Одним из способов, с помощью которого конвейеры процессоров решают эту проблему, является использование обходных путей. После завершения этапа выполнения первой инструкции (обозначенного как **1:В** на рис. 8.6), увеличенное на 1 значение в регистре eax было вычислено, но еще не было записано в регистр eax. Этапу выполнения второй инструкции (**2:В** на рис. 8.6) в качестве входных данных требуется увеличенное на 1 значение регистра eax. Если логика конвейера делает результат этапа 1:В доступным для этапа 2:В без предварительной записи в регистр eax, то выполнение второй инструкции можно завершить без задержки. Использование такого типа упрощения для передачи данных между исходной и целевой инструкциями без ожидания завершения исходной инструкции называется **обходом**. Обходы широко используются в архитектурах современных процессоров, чтобы обеспечить максимально эффективную работу конвейера.

В некоторых ситуациях обход невозможен, т. к. нужный результат просто не может быть вычислен до того, как целевая инструкция будет готова его использовать. В этом случае необходимо приостановить выполнение целевой инструкции и подождать доставки результата исходной инструкции. Это приводит к тому, что этап

выполнения конвейера простаивает, что обычно приводит к распространению периода простоя на остальные этапы конвейера.

Эта распространяющаяся по этапам задержка называется **конвейерным пузырем** по аналогии с воздушным пузырем, проходящим по трубопроводу с жидкостью. Наличие пузырей в последовательности обработки снижает эффективность конвейеризации.

Пузыри плохо влияют на производительность, поэтому разработчики конвейерных процессоров прилагают значительные усилия, чтобы по возможности избежать их. Одним из способов сделать это является **внеочередное выполнение инструкций**, обозначаемое **OoO** (Out-of-Order instruction execution). Рассмотрим две инструкции, упомянутые в предыдущем примере, но теперь за ними следует третья инструкция:

```
inc eax
add ebx, eax
mov ecx, edx
```

Третья инструкция не зависит от результатов выполнения предыдущих инструкций. Вместо использования обхода, чтобы избежать задержки, процессор может прибегнуть к внеочередному выполнению инструкций для исключения конвейерного пузыря. Итоговая последовательность выполнения инструкций будет выглядеть следующим образом:

```
inc eax
mov ecx, edx
add ebx, eax
```

Результат выполнения этих трех инструкций один и тот же, но разница во времени между первой и третьей инструкциями увеличилась, что снижает вероятность возникновения конвейерного пузыря, а если он все же возникнет, то такой порядок, по крайней мере, сокращает его продолжительность.

Метод внеочередного выполнения инструкций включает обнаружение зависимостей между инструкциями и перестраивает порядок их выполнения таким образом, чтобы получить те же общие результаты, но не в том порядке, в котором они были изначально закодированы. Некоторые процессоры (обычно CISC) выполняют это переупорядочивание инструкций в реальном времени во время выполнения программы. Другие архитектуры (обычно RISC) полагаются на интеллектуальные компиляторы языков программирования, которые осуществляют перестановку инструкций в процессе сборки программного обеспечения, чтобы свести к минимуму образование конвейерных пузырей. Первый подход требует значительных вложений в логику обработки и связанную с ней полезную площадь кристалла для выполнения переупорядочивания "на лету", в то время как второй подход упрощает

логику процессора, но существенно усложняет работу программистов на ассемблере, которые в этом случае несут ответственность за то, чтобы конвейерные пузыри не слишком ухудшали производительность выполнения программы.

В некоторых CISC-процессорах для реализации преимуществ по производительности, свойственных RISC-архитектурам, используются микрооперации и переименование регистров. Эти методы представлены в следующем разделе.

Микрооперации и переименование регистров

Архитектура набора инструкций *x*86 представляет особую проблему для разработчиков процессоров. Несмотря на то что история архитектуры *x*86 насчитывает уже несколько десятилетий, она остается основой персональных и коммерческих вычислений. Так как конфигурация CISC содержит всего восемь регистров (в 32-разрядной версии *x*86), методы высокоэффективной конвейеризации инструкций и использования большого количества регистров, доступных в архитектурах RISC, гораздо менее полезны в оригинальной архитектуре *x*86.

Для того чтобы воспользоваться некоторыми преимуществами методики RISC, разработчики процессоров *x*86 решили представить инструкции *x*86 в виде последовательностей микроопераций. Микрооперация, сокращенно **μop** (произносится как *"майкро-оп"*), является подэтапом инструкции процессора. Простые инструкции *x*86 разделяются на 1–3 микрооперации, а более сложные — не большее их количество. Разделение инструкций на микрооперации обеспечивает более высокий уровень детализации для оценки зависимостей от результатов других микроопераций и поддерживает увеличение параллелизма выполнения.

Вместе с поддержкой микроопераций современные процессоры обычно предоставляют дополнительные внутренние регистры (числом в десятки или даже сотни) для хранения промежуточных результатов микроопераций. Эти регистры содержат частично вычисленные результаты, предназначенные для переноса в физические регистры процессора после завершения всех микроопераций инструкции. Использование этих внутренних регистров называется **переименованием регистров**. Каждый переименованный регистр имеет соответствующее значение тега, указывающее, какой физический регистр процессора он в итоге займет. За счет увеличения количества переименованных регистров, доступных для промежуточного хранения, увеличиваются возможности параллелизма инструкций.

Несколько микроопераций могут находиться на разных стадиях выполнения в любой момент времени. Зависимости инструкции от результата предыдущей инструкции блокируют выполнение микрооперации до тех пор, пока не будут доступны ее входные данные, обычно передаваемые с помощью механизма обхода, описанного в предыдущем разделе. Как только все необходимые входные данные будут доступны, микрооперация будет запланирована для выполнения.

Этот режим работы представляет собой модель **обработки по потоку данных**. Такая обработка позволяет выполнять параллельные операции в суперскалярной ар-

хитектуре, инициируя выполнение микроопераций после разрешения любых зависимостей по данным.

Высокопроизводительные процессоры осуществляют декодирование инструкций в микрооперации после извлечения каждой инструкции. В некоторых микросхемах результаты этого декодирования хранятся в кеше инструкций уровня 0, расположенном между процессором и I-кешем L1. I-кеш L0 обеспечивает максимально быстрый доступ к предварительно декодированным микрооперациям для выполнения внутренних циклов кода с максимально возможной скоростью. Благодаря кешированию декодированных микроопераций процессор избегает необходимости повторять этапы конвейера, выполняющие декодирование инструкций, для каждого последующего доступа к одной и той же инструкции.

В дополнение к конфликтам, связанным с зависимостями по данным между инструкциями, вторым ключевым источником конфликтов конвейеризации является возникновение условного ветвления, обсуждаемого в следующем разделе.

Условное ветвление

Условное ветвление вносит существенные затруднения в процесс конвейерной обработки. Адрес инструкции, следующей за инструкцией условного ветвления, не может быть подтвержден до тех пор, пока не будет оценено условие ветвления.

Есть две возможности: если условие ветвления не выполняется, процессор выполнит инструкцию, которая следует за инструкцией условного ветвления. Если условие ветвления выполняется, следующая инструкция находится по адресу, указанному в инструкции.

Для решения проблем, связанных с условным ветвлением в конвейерных процессорах, используется несколько методов. Вот некоторые из них.

- Когда это возможно, следует вообще избегать ветвления. Разработчики программного обеспечения могут разрабатывать алгоритмы с внутренними циклами, которые сводят к минимуму или вовсе устраняют условный код. Оптимизирующие компиляторы стремятся перестроить и упростить последовательность операций, чтобы уменьшить негативные последствия условного ветвления.
- Процессор может отложить извлечение следующей инструкции до тех пор, пока не будет оценено условие ветвления. Обычно это приводит к образованию пузырей в конвейере, снижая производительность.
- Процессор может выполнить вычисление условия ветвления на как можно более ранних этапах конвейера. Благодаря этому можно быстрее идентифицировать правильную ветвь, обеспечивая выполнение с меньшей задержкой.
- Некоторые процессоры пытаются угадать результат условия ветвления и начинают предварительное выполнение инструкций по этому угаданному пути. Этот метод называется **прогнозированием ветвления**. Если предположение

оказывается неверным, процессор должен удалить результаты неправильно выполненных инструкций из конвейера (эта операция называется **очисткой конвейера**) и начать выполнение сначала — с первой инструкции по правильному пути. Неверное предположение приводит к значительному снижению производительности, однако правильное угадывание направления ветвления обеспечивает выполнение без задержек. Некоторые процессоры, в которых реализовано прогнозирование ветвления, отслеживают результаты оценок условия ветвления и используют эту информацию для повышения точности будущих прогнозов при повторном выполнении той же инструкции.

- При обнаружении инструкции условного ветвления процессор может начать выполнение инструкций по обоим путям ветвления одновременно, используя свои суперскалярные возможности. Этот подход называется **упреждающим выполнением**.

  После определения условия ветвления результаты выполнения по неправильному пути отбрасываются. Упреждающее выполнение может применяться только в том случае, если инструкции не вносят изменений, которые нельзя отменить. Запись данных в основную память или на устройство вывода является примером изменения, которое нелегко отменить, поэтому упреждающее выполнение будет приостановлено, если такое действие произойдет во время выполнения по неверному пути.

Современные высокопроизводительные процессоры выделяют значительную часть своих логических ресурсов для поддержки эффективной конвейеризации в самых разнообразных условиях обработки. Совокупные преимущества многоядерной, суперскалярной и суперконвейерной обработки обеспечивают впечатляющее повышение производительности в последних поколениях сложных процессорных архитектур. Реализуя конвейерную обработку инструкций и выполняя инструкции в суперскалярном контексте, эти функции привносят параллелизм в последовательности кода, изначально не предназначенные для параллельного выполнения.

Производительность обработки можно дополнительно повысить путем ввода параллельного выполнения нескольких потоков на одном ядре процессора с использованием одновременной многопоточности.

Одновременная многопоточность

Как мы узнали из предыдущих глав, каждый выполняемый процесс содержит один или несколько потоков выполнения. При реализации многопоточности с разделением по времени на одноядерном процессоре в любой момент времени в состоянии выполнения находится только один поток. Быстро переключаясь между несколькими готовыми к запуску потоками, процессор создает иллюзию (с точки зрения пользователя) одновременного выполнения нескольких программ.

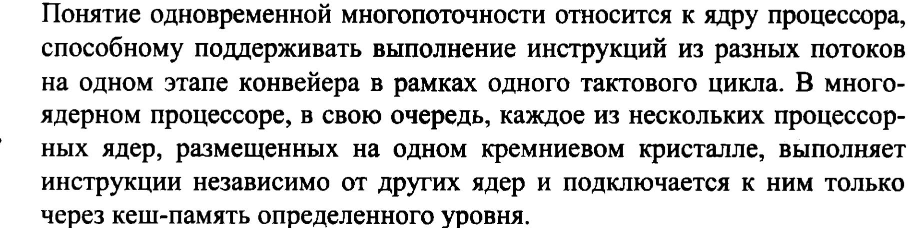
В этой главе представлена концепция суперскалярной обработки, которая предоставляет одному процессорному ядру возможность выдавать более одной инструк-

ции за такт. Повышение производительности в результате суперскалярной обработки может быть ограничено, когда активная последовательность инструкций не требует использовать сочетание ресурсов процессора, которое хорошо согласуется с возможностями его суперскалярных функциональных блоков. Например, в конкретной последовательности инструкций могут интенсивно использоваться блоки целочисленных вычислений (что приводит к появлению конвейерных пузырей), в то время как блоки вычисления адресов в основном простаивают.

Одним из способов увеличить использование суперскалярных возможностей процессора является выдача в каждом такте инструкций из более чем одного потока. Такой подход называют **одновременной многопоточностью**. Одновременное выполнение инструкций из разных потоков повышает вероятность того, что последовательности инструкций будут зависеть от несовместимых функциональных возможностей процессора, тем самым обеспечивая повышенный параллелизм выполнения и более высокую скорость выполнения инструкций.

Процессоры, поддерживающие одновременную многопоточность, должны предоставлять отдельный набор регистров для каждого одновременно выполняющегося потока, а также полный набор переименованных внутренних регистров для каждого потока. Цель здесь состоит в том, чтобы предоставить больше возможностей для использования суперскалярных функций процессора.

Многие современные высокопроизводительные процессоры поддерживают выполнение двух одновременных потоков, хотя некоторые обеспечивают поддержку до восьми таких потоков. Как и в случаях с большинством других методов повышения производительности, обсуждаемых в этой главе, увеличение числа одновременных потоков, поддерживаемых ядром процессора, в конечном счете достигает точки падения эффективности, т. к. одновременные потоки конкурируют за доступ к общим ресурсам.



ОДНОВРЕМЕННАЯ МНОГОПОТОЧНОСТЬ, МНОГОЯДЕРНЫЕ ПРОЦЕССОРЫ И МНОГОПРОЦЕССОРНОСТЬ



Понятие одновременной многопоточности относится к ядру процессора, способному поддерживать выполнение инструкций из разных потоков на одном этапе конвейера в рамках одного тактового цикла. В многоядерном процессоре, в свою очередь, каждое из нескольких процессорных ядер, размещенных на одном кремниевом кристалле, выполняет инструкции независимо от других ядер и подключается к ним только через кеш-память определенного уровня.

Многопроцессорный компьютер содержит несколько интегральных схем процессора, каждая из которых обычно входит в состав отдельного пакета интегральных схем. В качестве альтернативы многопроцессорный компьютер можно реализовать в виде нескольких микросхем процессоров, собранных в одном корпусе.

Обсуждавшиеся до сих пор методы оптимизации направлены на повышение производительности скалярной обработки данных. Это означает, что каждая последовательность инструкций работает с небольшим числом значений данных. Даже суперскалярная обработка, реализованная в сочетании с одновременной многопоточностью или без нее, направлена на ускорение выполнения инструкций, которые обычно в каждый момент времени работают с одним или двумя элементами данных размером с регистр.

При обработке векторных данных одна и та же математическая операция выполняется над каждым элементом массива значений данных одновременно. Особенности архитектуры процессора, улучшающие параллелизм выполнения операций векторной обработки, рассматриваются в следующем разделе.

Модель обработки SIMD

Процессоры, которые выдают за такт одну инструкцию без элементов данных или с одним или двумя элементами данных, называют скалярными процессорами. Процессоры, способные выполнять несколько инструкций за такт, но не выполняющие в явном виде инструкции векторной обработки, называются **суперскалярными процессорами**. Некоторые алгоритмы выигрывают от подхода к выполнению, явно ориентированного на векторную обработку, т. е. от одновременного выполнения одной и той же операции над многими элементами данных. Инструкции процессора, адаптированные к таким задачам, называют инструкциями, реализующими модель обработки **с одним потоком инструкций и множеством потоков данных** (single instruction, multiple data, SIMD).

Одновременно выдаваемые инструкции в суперскалярных процессорах обычно решают разные задачи с разными данными, представляя модель параллельной обработки **с множеством потоков инструкций и множеством потоков данных** (multiple instruction, multiple data, MIMD). Некоторые операции обработки, такие как используемая в цифровой обработке сигналов операция скалярного произведения, описанная в *главе 6*, выполняют одну и ту же математическую операцию над массивом значений.

Операция **умножения с накоплением** (multiply accumulate, MAC), также описанная в *главе 6*, последовательно выполняет скалярные математические операции над каждой парой элементов вектора. Также можно разработать схемные решения процессоров и инструкции, способные выполнять аналогичные операции более чем над одной парой чисел одновременно.

В современных процессорах предусмотрены SIMD-инструкции для выполнения таких задач, как математические операции с числовыми массивами, обработка графических данных и поиск подстрок в символьных строках.

Разработанные компанией Intel средства расширения потоковой обработки на основе модели SIMD — **Streaming SIMD Extensions (SSE)**, внедренные в процессорах Pentium III 1999 г., представляют собой набор инструкций и средств выполнения

для одновременной обработки 128-битных массивов данных. Данные, содержащиеся в массиве, могут состоять из целых чисел или значений с плавающей запятой. Инструкции SSE второго поколения (**SSE2**) могут параллельно обрабатывать данные следующих типов:

- два 64-битных значения с плавающей запятой;
- четыре 32-битных значения с плавающей запятой;
- два 64-битных целых значения;
- четыре 32-битных целых значения;
- восемь 16-битных целых значений;
- шестнадцать 8-битных целых значений.

SSE2 предоставляет инструкции вычислений с плавающей запятой для знакомых математических операций сложения, вычитания, умножения и деления.

Также доступны инструкции для вычисления квадратного корня, обратной величины, обратной величины квадратного корня и возврата максимального значения среди элементов массива. Инструкции целочисленных вычислений из SSE2 реализуют операции сравнения, побитовые операции, перетасовку данных и преобразование типов данных.

В более поздних поколениях набора инструкций SSE увеличены размер данных и разнообразие поддерживаемых операций. Последняя итерация возможностей типа SSE (по состоянию на 2021 г.) нашла выражение в виде набора расширений для усовершенствованных векторных вычислений AVX-512. **AVX** расшифровывается как **advanced vector extensions** и обеспечивает поддержку регистров разрядностью 512 бит. В состав AVX-512 включены, среди прочего, инструкции, оптимизированные для поддержки алгоритмов нейронных сетей.

Одно из препятствий на пути широкого внедрения различных поколений инструкций SSE и AVX заключается в том, что для их применения конечными пользователями эти инструкции должны быть реализованы в процессоре, операционная система должна поддерживать их, а компиляторы и другие аналитические инструменты для пользователей должны использовать возможности SSE. Исторически так сложилось, что после введения новых инструкций процессора требовались годы, прежде чем конечные пользователи могли легко использовать все их преимущества.

SIMD-инструкции, доступные в современных процессорах, по-видимому, нашли свое наиболее масштабное применение в области научных вычислений. Исследователям, работающим с комплексным моделированием, алгоритмами машинного обучения или сложным математическим анализом, наличие инструкций SSE и AVX дает возможность значительно повысить производительность кода, выполняющего обширные математические операции, манипуляции с символами и другие векторно-ориентированные задачи.

Резюме

Большинство современных 32- и 64-разрядных процессоров сочетают в себе значительную часть или даже все методы повышения производительности, представленные в этой главе. Типичный персональный компьютер или смартфон потребительского класса содержит единственную интегральную схему основного процессора с четырьмя ядрами, каждое из которых поддерживает одновременную многопоточность с двумя потоками. Этот процессор является суперскалярным, суперконвейерным и использует три уровня кеш-памяти. Процессор декодирует инструкции в микрооперации и выполняет сложное прогнозирование ветвления.

Методы, представленные в этой главе, могут показаться слишком сложными и трудными для понимания, однако в реальности каждый из нас регулярно применяет их и получает реализуемые ими преимущества в производительности каждый раз, когда мы пользуемся вычислительными устройствами любого типа. Логика обработки, необходимая для внедрения конвейеризации и суперскалярных операций, несомненно, сложна, но производители полупроводников прилагают усилия для реализации этих функций по одной простой причине: это окупается производительностью их изделий и итоговой ценностью этих продуктов в восприятии их клиентов.

В этой главе были представлены основные методы повышения производительности, используемые в процессорах и компьютерных архитектурах для достижения максимальной скорости выполнения в реальных вычислительных системах. Эти методы никоим образом не изменяют то, что процессор выдает на выходе; они просто помогают быстрее выполнять его работу. Мы рассмотрели наиболее важные методы повышения производительности систем, включая использование кеш-памяти, конвейерную обработку инструкций, одновременную многопоточность и модель обработки SIMD.

Следующая глава посвящена расширениям, обычно реализуемым на уровне набора инструкций процессора с целью предоставления системам дополнительных возможностей помимо общих требований к обработке данных.

Среди расширений, обсуждаемых в *главе 9*, — привилегированные режимы процессора, арифметика с плавающей запятой, управление питанием и управление безопасностью системы.

Упражнения

1. Рассмотрим I-кеш L1 с прямым отображением размером 32 Кбайт. Каждая строка кеша состоит из 64 байт, а системное адресное пространство составляет 4 Гбайт. Сколько битов отведено для тега этого кеша? Биты с какими номерами (бит 0 — наименее значимый бит) составляют адресное слово?
2. Рассмотрим 8-канальный наборно-ассоциативный кеш инструкций и данных L2 объемом 256 Кбайт с 64 байтами в каждой строке кеша. Сколько наборов используется в этом кеше?

3. Процессор имеет 4-этапный конвейер с максимальными задержками 0,8; 0,4; 0,6 и 0,3 нс на этапах 1–4 соответственно. Если первый этап заменить двумя этапами с максимальными задержками 0,5 и 0,3 нс, соответственно, насколько увеличится тактовая частота процессора в процентном выражении?



Специализированные расширения процессоров

В предыдущих главах мы обсуждали особенности вычислительных архитектур общего назначения и некоторые архитектурные специализации, предназначенные для учета требований определенных предметных областей. Эта глава посвящена расширениям, обычно реализуемым на уровне набора инструкций процессора и в других аппаратных средствах вычислительной системы с целью предоставления дополнительных возможностей помимо общих требований к обработке данных.

Прочитав эту главу, вы получите представление о назначении привилегированных режимов процессора и особенностях их работы в многопроцессорных и многопользовательских контекстах. Вы ознакомитесь с концепциями процессоров и наборов инструкций для вычислений с плавающей запятой, методами управления питанием в устройствах с аккумуляторным питанием, а также с функциями процессора и системы, повышающими уровень безопасности.

В этой главе мы обсудим следующие расширения процессора и системные функции:

- привилегированные режимы процессора;
- арифметика с плавающей запятой;
- управление питанием;
- управление безопасностью системы.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Привилегированные режимы процессора

Большинство операционных систем, работающих на 32- и 64-разрядных процессорах, управляют доступом к системным ресурсам, используя концепцию уровней привилегий. Основными причинами управления доступом таким образом являются повышение стабильности системы, предотвращение несанкционированного взаимодействия с системными аппаратными средствами и предотвращение несанкционированного доступа к данным.

Привилегированное выполнение повышает стабильность системы, гарантируя, что только доверенному коду разрешено выполнять инструкции, которые обеспечивают неограниченный доступ к ресурсам, таким как регистры конфигурации процессора и устройства ввода-вывода. Ядро операционной системы и связанные с ним модули, включая драйверы устройств, требуют привилегированного доступа для выполнения своих функций. Так как любой сбой процесса ядра или драйвера устройства, вероятнее всего, приведет к немедленной остановке работы всей системы, перед выпуском для общего использования эти программные компоненты обычно проходят процесс тщательного проектирования и подвергаются строгому тестированию.

Запуск операционной системы в привилегированном контексте предотвращает доступ неавторизованных приложений к управляемым системой структурам данных, таким как таблицы страниц и таблицы векторов прерываний. Случайно или в результате злого умысла пользовательское приложение может попытаться получить доступ к данным, находящимся в системной памяти или в памяти, принадлежащей другому пользователю. Механизмы управления доступом к системе предотвращают эти попытки и информируют неправильно работающее приложение о нарушении правил, инициируя исключение, обычно приводящее к завершению работы программы с сообщением об ошибке.

В *главе 3* были представлены концепции, связанные с прерываниями и их обработкой процессором. Прежде чем углубиться в детали привилегированных режимов процессора, мы обсудим обработку прерываний и исключений более подробно.

Обработка прерываний и исключений

Аппаратные прерывания, как мы видели, позволяют процессорам быстро реагировать на запросы обслуживания от периферийных устройств. Аппаратное прерывание уведомляет процессор о необходимости предпринять какое-либо действие, обычно связанное с передачей данных на внешнее устройство или от него.

Исключения аналогичны прерываниям с той лишь разницей, что исключения обычно связаны с реакцией на некоторые условия, возникающие внутри процессора. Одним из примеров внутреннего исключения является деление на ноль.

Код пользовательского уровня может намеренно генерировать исключения. На практике это стандартный метод, используемый непривилегированным кодом для

запроса системных сервисов, предоставляемых привилегированным кодом в ядре операционной системы и драйверах устройств.

Невозможно точно сформулировать различие между терминами "**прерывание**" и "**исключение**". Обработка прерываний и обработка исключений обычно зависят от одних и тех же или подобных аппаратных ресурсов процессора, работающих практически одинаково. В архитектуре *x*86 инструкция, которая инициирует программно-генерируемое прерывание (или исключение), обозначается как `int` — сокращение от *interrupt* (прерывание).

Исключения могут возникать в результате ошибок, возникающих во время выполнения программ, таких как деление на ноль, а также в ситуациях, не связанных с ошибками, таких как отказы страниц. Если прерывание или исключение не приводят к радикальной реакции в виде прекращения работы приложения, процедура обслуживания прерывания или обработчик исключений обрабатывает событие и возвращает управление системному планировщику, который в конечном счете возобновляет выполнение прерванного потока.

При возникновении прерывания или исключения процессор передает управление соответствующему обработчику следующим образом.

Когда происходит *прерывание*, процессор разрешает завершить выполнение текущей инструкции, а затем передает управление **процедуре обслуживания прерывания** (interrupt service routine, ISR). Код в ISR осуществляет обработку, необходимую для выполнения запроса, инициированного прерыванием. После завершения ISR выполнение прерванного кода возобновляется. Эти действия выполняются без ведома или подтверждения со стороны каких-либо потоков, которые, возможно, выполнялись во время прерывания.

При реагировании на *исключение* процессор передает управление процедуре обработки исключений, которая во многом похожа на ISR. Во время выполнения инструкции может возникнуть исключение, препятствующее завершению операций инструкции. Если выполнение инструкции прерывается из-за исключения, та же инструкция будет перезапущена после завершения работы обработчика исключений и возобновления выполнения потока. Этот механизм допускает возникновение отказов страниц в любой момент во время выполнения программы без влияния на результаты, полученные программой (кроме введения задержки из-за обработки отказа страницы).

Каждый тип прерывания или исключения имеет номер вектора, ссылающийся на строку в таблице векторов системных прерываний. При возникновении прерывания или исключения аппаратные средства процессора обращаются к **таблице векторов прерываний** (interrupt vector table, IVT) для выбора соответствующей процедуры обслуживания прерывания (ISR) или обработчика исключений. Строка таблицы векторов прерываний, соответствующая номеру вектора, содержит адрес процедуры обслуживания прерывания или обработчика для этого вектора.

При обработке прерывания или исключения процессор помещает любую необходимую контекстную информацию в стек и передает управление обработчику. В случае прерывания обработчик обслуживает устройство и удаляет его запрос на

прерывание. После обработки прерывания или исключения обработчик возвращает управление операционной системе. В случае исключения это может повлечь за собой перезапуск инструкции, которая выполнялась в момент возникновения исключения.

В табл. 9.1 приведены некоторые типы прерываний и исключений, используемых в процессорах *x*86, работающих в защищенном режиме.

Таблица 9.1. *Пример типов прерываний и исключений в x86*

Вектор	Описание	Причина
0	Ошибка деления	Инструкция div или idiv предприняла попытку целочисленного деления на ноль
2	Немаскируемое прерывание	Был получен аппаратный сигнал NMI
3	Контрольная точка	Выполнена инструкция int 3
6	Недопустимый код операции	Была предпринята попытка выполнить зарезервированный код операции
13	Общая защита	Была предпринята попытка запрещенного доступа к памяти или системному ресурсу
14	Отказ страницы	MMU выдал запрос на трансляцию виртуального адреса
32–255	Определяются пользователем	Эти векторы доступны для использования аппаратными прерываниями или с помощью инструкции int

Отметим некоторые интересные детали, связанные с прерываниями и исключениями, указанными в табл. 9.1.

- Работа входного сигнала NMI по вектору 2 аналогична входу NMI процессора 6502, рассмотренному в *главе 3*.
- Современные процессоры предоставляют развитые возможности использования контрольных точек, не нарушающие работу системы, однако архитектура *x*86 сохраняет функциональность контрольной точки, предоставляемую инструкцией int 3 с самых первых дней существования процессора 8086. Как было показано в *главе 3*, механизм, используемый отладчиками программного обеспечения процессора 6502 для прерывания выполнения программы по указанному адресу, заключается во временной замене кода операции в месте прерывания на опкод BRK. Когда выполнение программы достигает этого места, обработчик BRK берет на себя управление, чтобы предоставить пользователю возможность взаимодействовать с системой. Инструкция int 3 в архитектуре *x*86 функционирует таким же образом.
- В отличие от инструкции int процессоров *x*86, используемой с любыми другими номерами векторов, инструкция int 3 реализована в виде однобайтового опкода со значением CCh. Программное прерывание с другим вектором, на-

пример `int 32`, представляет собой двухбайтовую инструкцию. Инструкция `int 3` позволяет вставлять контрольную точку путем замены одного байта кода значением CCh.

- Вектор 13, обработчик исключений общей защиты, активируется при любой попытке приложения получить доступ к памяти или системному ресурсу, не выделенному для такого использования. В современных операционных системах системный обработчик для этого вектора завершает работу запущенного приложения и отображает сообщение об ошибке.
- Вектор 14, обработчик отказов страниц, срабатывает, когда приложение пытается получить доступ к допустимой странице, которая отсутствует в физической памяти. Этот обработчик пытается найти указанную страницу, которая может находиться в файле на диске или в системном файле подкачки, загружает страницу в память, обновляет таблицы трансляции виртуальных адресов в физические и перезапускает инструкцию, которая вызвала исключение.

Подводя итог, можно сказать, что аппаратные прерывания инициируются устройствами ввода-вывода, нуждающимися в передаче данных или других видах обслуживания. Исключения возникают, когда выполнение последовательности инструкций должно быть приостановлено для обработки такого условия, как отказ страницы или попытка несанкционированного доступа к памяти.

Аппаратные прерывания, как правило, происходят в случайные моменты времени относительно текущего выполнения кода программы, в то время как поведение, связанное с исключениями, генерируемыми программным образом, часто повторяется, если тот же код выполняется снова при работе с теми же данными.

Некоторые исключения, такие как ошибка общей защиты, приводят к завершению процесса, вызвавшего исключение, но большинство прерываний и исключений заканчиваются возобновлением выполнения прерванного потока после того, как обработчик прерывания или исключения завершил свою работу.

ИСКЛЮЧЕНИЯ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Многие языки программирования предоставляют средства для обработки исключений в приложениях. Такие исключения существенно отличаются от исключений, обрабатываемых на уровне процессора. Исключения в языках программирования обычно связаны с ошибками на гораздо более высоком концептуальном уровне, чем исключения, обрабатываемые аппаратными средствами процессора.

Например, C++ генерирует (или "выдает", в терминологии C++) исключение при неудачном запросе на выделение памяти. Это исключение совсем другого типа по сравнению с системными исключениями, обрабатываемыми на уровне процессора. Будьте внимательны и не путайте исключения в языках программирования высокого уровня с исключениями, обрабатываемыми непосредственно процессором.

Кольца защиты

Стратегия защиты, применяемая в современных процессорах и операционных системах, в чем-то похожа на систему обороны, нашедшую воплощение в устройстве средневековых замков. За́мок обычно имеет высокую стену, окружающую его территорию, иногда дополненную рвом. В этой внешней стене имеется небольшое количество хорошо защищенных проходов, каждый из которых защищен от злоумышленников такими механизмами, как разводные мосты и круглосуточная охрана. Сам за́мок, расположенный на огороженной стеной территории, также имеет крепкие стены и небольшое количество хорошо защищенных проходов, что еще больше ограничивает доступ к наиболее уязвимым местам.

Самые привилегированные члены за́мковой иерархии имеют неограниченные права на вход и выход из замка и проход через внешнюю стену. Менее привилегированные члены общества могут иметь разрешение на проход внутрь через внешнюю стену, но им запрещен доступ непосредственно в замок. Наименее привилегированным представителям местного населения в большинстве случаев запрещено входить в замок, а в тех случаях, когда доступ им разрешен (например, в определенные общественные зоны), они должны принять ограничения в отношении того, что им позволено делать.

Защиту, обеспечиваемую этой стратегией, можно представить в виде концентрических колец, при этом для доступа к внутренним кольцам требуются самые высокие привилегии, а внешнее кольцо представляет собой область, требующую наименьших привилегий, как показано на рис. 9.1.



Рис. 9.1. *Пример защитных колец*

Эта стратегия защиты реализует три уровня привилегий, которые определяют типы доступа для всех пользователей системы. Кольцо 0 требует самого высокого уровня привилегий, а кольцо 2 не требует особых привилегий.

Современные процессоры и работающие на их основе операционные системы используют очень похожий подход для предотвращения несанкционированного доступа к критическим ресурсам и предоставления непривилегированным пользователям доступа к возможностям системы, одобренным для их использования.

В принципе, можно создать несколько промежуточных уровней между кольцами с наивысшими и низшими привилегиями, однако в большинстве современных компьютерных систем реализованы только два кольца: привилегированное кольцо, называемое ядром или супервизором, и непривилегированное пользовательское кольцо. В некоторых операционных системах имеется промежуточное кольцо, которое содержит драйверы устройств и предоставляет доступ к ресурсам, необходимым для взаимодействия с устройствами ввода-вывода, но не обеспечивает беспрепятственного общесистемного доступа в кольцо ядра.

Одна из причин, по которой операционные системы, такие как Windows и Linux, поддерживают только два кольца привилегий, заключается в том, что при проектировании этих систем ставилась задача переносимости на различные процессорные архитектуры. Некоторые процессоры поддерживают только два кольца привилегий, но есть и другие, которые поддерживают большее количество колец. Переносимая операционная система не может реализовать более двух колец привилегий, если архитектура какого-либо из базовых процессоров не поддерживает желаемое количество колец. Архитектура *x*86, например, поддерживает до четырех колец, но в Windows и Linux используются только два из них (кольцо 0 с наибольшими привилегиями и кольцо 3 с наименьшими привилегиями).

На рис. 8.2 представлена кольцевая организация большинства операционных систем, работающих на архитектуре *x*86.



Рис. 9.2. *Кольца защиты в операционных системах x86*

Кольцевая система управления привилегиями является основной причиной того, что печально известный "синий экран смерти" Windows, который был распростра-

нен в 1990-х годах, так редко появляется у пользователей последних версий Windows. Пользовательские приложения, такие как веб-браузеры, почтовые клиенты и текстовые редакторы, иногда сталкиваются с проблемами, которые приводят к аварийному завершению их работы. Благодаря двухуровневому механизму применения привилегий, предоставляемому современными операционными системами, такими как Windows, Linux, macOS и Android, ущерб от сбоя в работе отдельного приложения сдерживается операционной системой, не позволяя ему повлиять на саму операционную систему или любую другую программу, запущенную в системе (по крайней мере, на те программы, которые не зависят от корректной работы программы, в которой произошел сбой).

После аварийного завершения работы приложения операционная система очищает все ресурсы, использовавшиеся отказавшей программой, включая выделенные страницы памяти и открытые файлы. Благодаря этому компьютерные системы, такие как веб-серверы, могут непрерывно работать в течение сотен дней, несмотря на периодические сбои и перезапуски работающих на них программных приложений.

Помимо защиты от сбоев приложений, управление привилегиями на основе колец дает существенные преимущества в области защиты от злоумышленников. Один из типов атак, которые может попытаться предпринять хакер, — вставить некоторый код в системный модуль, работающий в кольце 0. Такая вставка может произойти в исполняемый файл на диске, или злоумышленник может попытаться изменить код ядра во время его работы в памяти. В случае успеха злоумышленник может использовать этот код для доступа к данным в любой части системы, поскольку он работает в кольце 0.

Успех такой атаки не является чем-то невозможным, однако в современных процессорах и операционных системах реализован обширный набор мер безопасности и устранены многие уязвимости, присутствовавшие в ранних версиях операционных систем. Когда системные администраторы и пользователи в полной мере используют преимущества средств защиты на основе колец, доступных в современных компьютерных системах, у злоумышленника остается очень мало возможных путей для получения доступа к защищенным данным. Действительно, в большинстве случаев ключевым элементом успешных взломов, о которых сообщают в наши дни в новостях, является нарушение правил безопасности человеком, которое использовал в своих целях злоумышленник. Мы обсудим киберугрозы и методы обеспечения безопасности системного и прикладного программного обеспечения позже в *разд. "Управление безопасностью системы" этой главы*, а также в *главе 14*.

Режим супервизора и режим пользователя

В двухуровневой иерархии защиты на основе колец уровень защиты выполняемого потока представлен битом в реестре. При работе в кольце 0 бит **режима супервизора** имеет значение 1, а при работе в **режиме пользователя** (кольцо 3 в *x*86) этот бит равен 0. Бит режима супервизора может быть изменен только кодом, работающим в режиме супервизора.

Состояние этого бита определяет, какие инструкции доступны для выполнения потоком. Инструкции, которые могут помешать работе системы, например инструкция `hlt` в *x*86, которая останавливает выполнение инструкций процессора, недоступны в режиме пользователя.

Любая попытка выполнить запрещенную инструкцию приводит к сбою общей защиты. В режиме пользователя доступ инструкций к областям системной памяти и памяти, выделенной другим пользователям, запрещен. В режиме супервизора все инструкции доступны для выполнения, и также доступны все области памяти.

В аналогии с за́мком бит режима супервизора представляет собой предоставляемое охранникам замка разрешение, открывающее доступ на территорию замка и в сам замок. Установленный бит режима супервизора предоставляет вам ключи от королевства.

Системные вызовы

Весь код, относящийся к ядру и драйверам устройств, всегда работает в кольце 0. Весь пользовательский код всегда выполняется в кольце 3, даже для пользователей с повышенными привилегиями в операционной системе, таких как системные администраторы. Код, выполняющийся в кольце 3, строго контролируется системой и не может напрямую выполнять какие-либо действия, связанные с выделением памяти, открытием файла, выводом информации на экран или взаимодействием с устройством ввода-вывода. Для того чтобы получить доступ к любой из этих системных функций, пользовательский код кольца 3 должен отправить в ядро запрос на обслуживание.

Этот запрос на обслуживание должен сначала пройти через "ворота" (как посетители, входящие в наш замок), где тип запрашиваемой операции, а также все связанные с ней параметры проходят тщательную проверку и утверждение, прежде чем будет выдано разрешение на выполнение операции. Выполняющий запрошенную операцию код запускается в режиме супервизора в кольце 0 и по завершении возвращает управление в режим пользователя в кольце 3.

В ранних версиях Windows (до Windows XP) для запроса системных услуг приложение использовало механизм программного прерывания с вектором `2eh`. Протокол вызова системного сервиса включал в себя помещение в регистры процессора параметров, необходимых для запрашиваемого сервиса, и выполнение инструкции `int 2eh`, инициирующей программное прерывание. Обработчик работал в режиме супервизора, что приводило к переходу от кольца 3 к кольцу 0. После завершения работы обработчика система возвращалась к кольцу 3 для выполнения инструкции, следующей за `int 2eh`.

Одна из проблем, связанных с использованием механизма int 2eh для запроса услуг ядра, заключается в его невысокой эффективности. Действительно, требуется более 1000 тактовых циклов процессора, чтобы пройти путь от точки, в которой выполняется инструкция int 2eh, до кода ядра, который начинает обрабатывать это исключение. Загруженная система может запрашивать услуги ядра тысячи раз в секунду.

Для решения этой проблемы Intel внедрила в архитектуре *x*86, начиная с процессора Pentium II в 1997 г., инструкции sysenter и sysexit. Их цель заключалась в ускорении процесса перехода из кольца 3 в кольцо 0, а затем возврата в кольцо 3. За счет использования этих инструкций вместо int 2eh процесс перехода в режим ядра и возвращение из него ускоряется примерно в три раза.

Примерно в то время, когда компания Intel начала выпускать процессоры с инструкциями sysenter и sysexit, AMD выпустила инструкции syscall и sysret в своих процессорных архитектурах с той же целью повышения производительности. К сожалению, инструкции в процессорных архитектурах Intel и AMD несовместимы, в результате чего к операционным системам предъявляется требование различать архитектуры при использовании инструкций для ускорения вызова сервисов ядра.

В следующем разделе мы рассмотрим форматы данных и операции, связанные с арифметикой с плавающей запятой.

Арифметика с плавающей запятой

Современные процессоры обычно поддерживают целочисленные типы данных разрядностью 8, 16, 32 и 64 бита. Некоторые небольшие встраиваемые процессоры могут не поддерживать 64-битные или даже 32-битные целые числа напрямую, в то время как более сложные устройства способны поддерживать 128-битные целые числа. Целочисленные типы данных подходят для использования в широком спектре приложений, но многие области вычислений, особенно в сфере науки, инженерного дела и навигации, требуют возможности с высокой степенью точности представлять дробные числа.

В качестве простого примера ограничений целочисленной математики предположим, что вам нужно разделить 5 на 3. На компьютере, ограниченном использованием только целых чисел, вы можете выполнить целочисленное вычисление этого выражения, используя следующий код на языке C++:

```
#include <iostream>
int main(void)
{
    int a = 5;
    int b = 3;
    int c = a / b;
    std::cout << "c = " << c << std::endl;
    return 0;
}
```

Эта программа выдает следующий результат:

```
c = 1
```

Если вы введете это выражение в карманный калькулятор, то обнаружите, что результат, полученный программой, не слишком близок к фактическому результату, который составляет примерно 1,6667. В вычислительных приложениях, где требуются точные вычисления с действительными числами, практическое решение предлагает арифметика с плавающей запятой.

С точки зрения математики, множество действительных чисел состоит из всех чисел, включая все целые и дробные числа, расположенные на числовой прямой от отрицательной бесконечности до положительной бесконечности. Количество цифр в целой и дробной частях действительного числа не ограничено.

Учитывая ограниченный объем памяти, доступный даже в самом большом компьютере, очевидно, что в компьютерных программах невозможно представить весь диапазон действительных чисел. Если мы хотим представлять действительные числа полезным образом, необходим компромисс. Стандартным решением является представление действительных чисел с помощью мантиссы и порядка.

В областях знаний, связанных с математикой очень больших или очень малых чисел, принято представлять такие числа в виде мантиссы и порядка. Например, в физике гравитационная постоянная представлена в следующем формате:

$$G = 6{,}674 \times 10^{-11}\ \frac{\text{м}^3}{\text{кг} \cdot \text{с}^2}.$$

Мантисса представляет собой ненулевые цифры числа после умножения на масштабный коэффициент, который помещает эти цифры в удобный диапазон. **Порядок** представляет собой масштабный коэффициент, на который необходимо умножить мантиссу для получения фактического значения.

В этом примере мантисса составляет 6,674, а порядок равен –11. В данном примере используется десятичная мантисса, что удобно для ручных расчетов, поскольку умножение на масштабный коэффициент 10^{-11} можно выполнить, просто переместив положение десятичной запятой мантиссы. В этом примере порядок –11 требует перемещения десятичной запятой на 11 мест влево, что дает эквивалентное значение $G = 0{,}00000000006674$.

Применение нотации с плавающей запятой позволяет избежать ошибок, связанных с использованием длинных последовательностей нулей, и дает возможность представлять как очень большие, так и очень малые числа в компактном формате.

В нотации с плавающей запятой можно представить любое число на линии действительных чисел. Однако, если целью является представление всех действительных чисел, не должно быть ограничений по количеству цифр в мантиссе или порядке.

В компьютерных представлениях чисел с плавающей запятой и мантисса, и порядок ограничены предопределенной шириной битового поля. Эти диапазоны выбираются таким образом, чтобы числа с плавающей запятой укладывались в стандартную разрядность данных, обычно 32 или 64 бита, обеспечивая при этом достаточное количество разрядов мантиссы и порядка для широкого спектра практических задач.

Увеличение числа разрядов мантиссы повышает точность числовых значений, воспроизводимых в формате с плавающей запятой. Увеличение числа цифр порядка увеличивает диапазон чисел, которые могут быть представлены. Из-за конечной длины мантиссы и порядка результат вычислений с плавающей запятой часто не соответствует реальному результату. Вместо этого лучший исход в такой ситуации заключается в том, что результат с плавающей запятой будет ближайшим представимым значением к правильному результату.

Современные процессоры обычно поддерживают 32- и 64-битное представление чисел с плавающей запятой. Вместо десятичного порядка, описанного в предыдущем примере, компьютеры работают с порядком по основанию 2. Общий формат компьютерного числа с плавающей запятой:

$$\text{знак} \times \text{мантисса} \times 2^{\text{порядок}}.$$

Знак — это просто +1 или –1. В двоичном представлении положительное число имеет знаковый бит, равный 0, а отрицательное число — знаковый бит, равный 1. Если отделить знак от остальной части числа, оставшееся значение является неотрицательным. Для любого значения, отличного от нуля (который рассматривается как особый случай), это число можно привести в диапазон от 1 (включительно) до менее 2 путем умножения на некоторую степень 2.

Продолжая пример с гравитационной постоянной, поскольку знак равен +1, значение после удаления знака остается неизменным: $6{,}674 \times 10^{-11}$. Мантиссу этого числа можно привести в диапазон ($1 \le m < 2$), где m представляет собой мантиссу, умноженную на 2^{34}. Результат этого масштабирования:

$$G = +1 \times 1{,}1465845 \times 2^{-34}.$$

Для того чтобы мантисса попала в нужный диапазон, мы умножили исходное число на 2^{34}, поэтому, чтобы компенсировать операцию масштабирования, представление числа с плавающей запятой необходимо умножить на 2^{-34}.

Так как наши компьютеры работают с двоичными числами, мы должны преобразовать мантиссу в двоичное представление. Формат, используемый при обработке с плавающей запятой, должен представлять число из диапазона от 1 до 2 в виде целого числа без знака.

Например, если мы предположим, что двоичная мантисса имеет разрядность 16 бит, то мантисса 1,0 будет представлена числом `0000h`, а значение чуть меньше 2,0 (фактически, 1,99998474) — числом `FFFFh`. Десятичная мантисса m преобразуется в двоичную 16-битную мантиссу с помощью выражения $(m - 1) \times 2^{16}$ и округления результата до ближайшего целого числа. Двоичная 16-битная мантисса m преобразуется в десятичную с помощью выражения $1 + m \times 2^{-16}$.

В нашем примере десятичная мантисса 1,1465845 преобразуется в 16-битную двоичную мантиссу вида `0010010110000111b` или `2587h`.

С помощью процедуры масштабирования, описанной в этом разделе, числа с плавающей запятой можно представить в двоичной форме с любой желаемой разрядностью для мантиссы и порядка.

Для целей совместимости между процессорами полезно определить ограниченное число двоичных форматов с плавающей запятой, способных поддерживать широкий спектр вариантов применения, и принять эти форматы в масштабах всей отрасли.

По этой причине в 1985 г. был принят стандарт IEEE 754, определяющий компьютерное представление чисел с плавающей запятой. Прежде чем ознакомиться с основными положениями этого стандарта, рассмотрим конструкцию сопроцессора Intel 8087, предназначенного для вычислений с плавающей запятой, который послужил источником многих концепций, позднее закрепленных в стандарте IEEE 754.

Сопроцессор 8087 для вычислений с плавающей запятой

Современные процессоры с аппаратной поддержкой вычислений с плавающей запятой обычно реализуют набор соответствующих инструкций и выполняют эти операции в выделенном функциональном блоке. В суперскалярном процессоре ядро главного процессора выполняет инструкции других категорий, а математический сопроцессор (floating point unit, FPU) параллельно выполняет вычисления с плавающей запятой.

Вспомните *главу 1*, там говорилось, что оригинальный IBM PC 1981 г. содержал гнездо для сопроцессора 8087 с плавающей запятой. Аппаратные средства сопроцессора 8087 выполняют вычисления с плавающей запятой со скоростью примерно в 100 раз выше, чем функционально эквивалентная ему программная реализация, работающая на главном процессоре.

Поскольку установка 8087 была необязательной, большинство программ для ПК, которые хотели воспользоваться его возможностями, сначала проверяли наличие 8087 и в случае его отсутствия возвращались к библиотеке гораздо более медленного кода для вычислений с плавающей запятой.

Сопроцессор 8087 поддерживает следующие типы данных для обработки чисел:

- 16-битное целое число в дополнительном коде;
- 32-битное целое число в дополнительном коде;
- 64-битное целое число в дополнительном коде;
- 18-значное упакованное **двоично-десятичное число** (binary coded decimal, BCD) со знаком;
- 32-битное короткое действительное число со знаком, 24-битной мантиссой и 8-битным порядком;
- 64-битное длинное действительное число со знаком, 53-битной мантиссой и 11-битным порядком;
- 80-битное временное действительное число со знаком, 64-битной мантиссой и 15-битным порядком.

Данные каждого типа хранятся в памяти в виде последовательности байтов. Форматы, используемые для типов данных, представляющих действительные числа, показаны на рис. 9.3.



Рис. 9.3. *Форматы данных с плавающей запятой в сопроцессоре 8087*

Форматы коротких и длинных действительных чисел используют подразумеваемый бит 1 в качестве старшего бита мантиссы и не включают этот бит в свои двоичные представления. В качестве особого случая ноль представляется в этих форматах путем установки нулевых значений для мантиссы и порядка.

Формат временного действительного числа используется внутри сопроцессора 8087 для хранения промежуточных результатов. Этот формат имеет увеличенную точность по сравнению с форматом длинного действительного числа, чтобы свести к минимуму распространение ошибок округления по серии вычислений.

Каждый формат действительных чисел можно представить в виде

$$(-1)^S(2^{E\,-\,\text{смещение}})(m),$$

где S — это бит знака, E — порядок, а m — мантисса. *Смещение* равно 127 для формата коротких действительных чисел, 1023 для формата длинных действительных чисел и 16 383 для формата временных действительных чисел.

Вычитание *смещения* преобразует беззнаковое целое число, хранящееся в поле порядка, в знаковое значение.

В нашем примере действительное число G с десятичной мантиссой 1,1465845 и двоичным масштабом 2^{34} представлено в формате короткого действительного числа со знаковым битом, равным нулю, порядком $(-34 + 127) = 5Dh$ и мантиссой $(1{,}1465845 - 1) \times 2^{23} = 12C348h$. Комбинируя все три компонента, получаем следующее 32-битное представление значения:

$$6{,}674 \times 10^{-11} : (\text{бит знака} = 0) \times 2^{31} + \textit{смещенный порядок} =$$
$$= 5Dh \times 2^{23} + (\text{мантисса} = 12C348h).$$

Итоговое 32-битное значение с плавающей запятой одинарной точности: 2E92C348h.

Сопроцессор 8087 добавляет к набору инструкций 8086/8088 68 обозначений опкодов для выполнения арифметических, тригонометрических, экспоненциальных и логарифмических функций. В программе для ПК, использующей сопроцессор 8087, код состоит из одного потока инструкций 8088 и 8087, извлекаемых обычным по-

следовательным способом процессором 8088. Сопроцессор 8087 пассивно контролирует шины адресов и данных по мере выполнения процессором инструкций и подключается только при появлении опкода 8087. Процессор 8088 рассматривает инструкции 8087 как инструкции без операции (или nop) и игнорирует их.

Когда 8087 начинает выполнять инструкцию, он может взять под контроль шину хоста для передачи данных между своими внутренними регистрами и системной памятью с помощью циклов прямого доступа к памяти (DMA). 8087 и 8088 не передают данные между собой напрямую и могут обмениваться данными, только сохраняя их в памяти для использования другим процессором.

Выполнение инструкций сопроцессора 8087 происходит независимо от 8088, что делает ПК, оснащенный 8087, по-настоящему параллельной системой обработки. В 8087 предусмотрен выходной сигнал BUSY, позволяющий процессору 8088 определить, обрабатывает ли 8087 в данный момент инструкцию.

Когда процессору 8088 требуются результаты работы 8087, 8088 должен подождать отключения сигнала BUSY, и в этот момент он получает доступ к ячейкам памяти, содержащим результат инструкции 8087.

Стандарт вычислений с плавающей запятой IEEE 754

Наиболее широко используемыми форматами представления чисел с плавающей запятой в современных компьютерных системах являются форматы, определенные в стандарте IEEE 754. **Институт инженеров по электротехнике и электронике** (Institute of Electrical and Electronic Engineers, IEEE) публикует широкий спектр стандартов, связанных с электротехникой, электроникой и вычислительной техникой. Первоначальная версия стандарта IEEE 754 была принята в 1985 г. и в значительной степени основывалась на типах данных и математических операциях сопроцессора Intel 8087.

8087 не полностью соответствовал первоначальной версии стандарта IEEE 754, который был опубликован через несколько лет после появления 8087. Более поздние сопроцессоры Intel с плавающей запятой, начиная с 80387 в 1987 г., уже полностью соответствовали этому стандарту. В современных 32- и 64-разрядных процессорах блок вычислений с плавающей запятой, соответствующий стандарту IEEE 754, обычно размещают на той же интегральной микросхеме, что и основной процессор.

Стандарт IEEE 754 был обновлен в 2008 г. и еще раз — в 2019 г. Текущая версия IEEE 754-2019 содержит определения для форматов чисел с плавающей запятой разрядностью 16, 32, 64, 128 и 256 бит. Эта версия также содержит форматы десятичных чисел с плавающей запятой разрядностью 32, 64 и 128 бит. 32-битный и 64-битный форматы с плавающей запятой для двоичных чисел обычно поддерживаются в языках программирования, включающих операции с плавающей запятой. Поддержка остальных типов данных IEEE 754, как правило, имеет более ограниченное распространение и не является стандартной для разных процессоров, языков программирования и операционных систем.

В следующем разделе представлены функции, реализованные во многих современных процессорных архитектурах для управления энергопотреблением системы.

Управление питанием

Для пользователей мобильных устройств с питанием от аккумулятора, таких как смартфоны, планшеты и ноутбуки, возможность длительной работы без подзарядки является важной характеристикой. Разработчики мобильных устройств уделяют большое внимание тому, чтобы расход энергии аккумулятора был минимальным при любых условиях эксплуатации.

Вот некоторые методы, используемые проектировщиками для снижения энергопотребления.

- Перевод неработающих подсистем в состояние низкого энергопотребления или их полное отключение, когда они не нужны. Применение этого метода может оказаться невозможным для периферийных устройств, которые должны быть доступны для реагирования на входящие запросы, например от сетевых интерфейсов.
- Снижение напряжения питания интегральных схем и тактовых частот в периоды, когда скорость выполнения не критична.
- Когда это возможно, сохранение информации о состоянии системы и выключение питания процессора. Пользователи мобильных компьютеров знакомы с двумя режимами снижения энергопотребления, когда система не используется: режим ожидания и спящий режим. В режиме ожидания питание подается только на оперативную память системы, в то время как остальная часть системы выключена. Режим ожидания обеспечивает очень быстрый запуск, когда пользователь возобновляет работу с системой. За эту оперативность приходится платить: для поддержания оперативной памяти в рабочем состоянии требуется значительное количество энергии.

  В спящем режиме все данные о состоянии системы записываются на диск, и система полностью выключается. Спящий режим не требует практически никакого питания, хотя для возобновления работы обычно нужно немного больше времени, чем в режиме ожидания.
- Если требуется периодическая обработка, например в приложениях реального времени, то при завершении каждого этапа обработки процессор переводится в состояние низкого энергопотребления. Процессор остается в этом состоянии до тех пор, пока прерывание по таймеру (или прерывание другого типа) не "разбудит" процессор для следующей итерации. Многие встраиваемые процессоры поддерживают режим ожидания с низким энергопотреблением, в котором конвейер инструкций останавливается, но остается готовым мгновенно возобновить выполнение в ответ на прерывание. Некоторые реализации ОСРВ поддерживают эту концепцию, переводя процессор в режим ожидания, когда все задачи блокируются в ожидании прерывания или другого события.

Смартфоны и другие устройства с питанием от аккумулятора широко используют эти методы для минимизации разряда аккумулятора, сохраняя готовность мгновенно отреагировать на действия пользователя и на внешние воздействия, такие как входящие вызовы и уведомления из социальных сетей.

Современные процессоры в высокопроизводительных компьютерах и мобильных устройствах обычно поддерживают возможность регулирования тактовой частоты процессора во время выполнения кода, а в некоторых случаях могут изменять напряжение питания процессора. Это сочетание методов управления питанием рассматривается в следующем разделе.

Динамическое изменение напряжения и частоты

Оптимизация тактовой частоты процессора и напряжения питания с целью свести энергопотребление к минимуму называется **динамическим изменением напряжения и частоты** (dynamic voltage frequency scaling, DVFS). В режиме DVFS процессор регулярно оценивает требования своего текущего рабочего состояния к производительности и регулирует тактовую частоту и напряжение питания, чтобы свести к минимуму использование заряда аккумулятора, продолжая при этом работать на приемлемом уровне.

В КМОП-чипе, содержащем множество транзисторов, разумная оценка потребляемой мощности составляет:

$$P = \sum_{i=1}^{N} C_i v_i^2 f_i.$$

В этом выражении P отражает общее энергопотребление чипа, который содержит N МОП-транзисторов; C_i — емкость, создаваемая i-м транзистором; v_i — напряжение питания этого транзистора; f_i — его рабочая частота.

Рассматривая эту формулу, мы видим, что потребляемая мощность схемы пропорциональна квадрату напряжения питания и прямо пропорциональна емкости и рабочей частоте. Поскольку емкость схемы играет значительную роль в энергопотреблении, выгодно производить устройство с помощью процесса изготовления, который уменьшает размер вентилей схемы, тем самым сводя емкость к минимуму. Это одна из причин, по которой переход на технологический процесс производства интегральных схем, поддерживающий меньшие размеры элементов, приводит к созданию устройств с пониженным энергопотреблением.

Режим DVFS пытается свести к минимуму общее энергопотребление, постоянно снижая напряжение питания и тактовую частоту процессора настолько, насколько это возможно.

Снижение напряжения питания схемы может обеспечить значительное снижение энергопотребления. Однако любое снижение напряжения питания процессора должно быть тщательно согласовано с изменением тактовой частоты. При снижении напряжения питания КМОП-транзисторы переключаются медленнее. Это обусловлено тем, что емкость, создаваемая каждым транзистором, остается неизмен-

ной, но ее заряд и разряд выполняются медленнее, т. к. процесс заряда/разряда происходит при меньшем напряжении. Для того чтобы при снижении напряжения системы схема работала правильно, необходимо одновременно снижать тактовую частоту.

По аналогии с гидравлической системой из *главы 2*, эффект от снижения напряжения питания КМОП эквивалентен снижению давления воды, наполняющей воздушный шарик, прикрепленный к стенке трубы: при снижении давления в системе шарик наполняется медленнее.

Снижение напряжения питания процессора также снижает помехоустойчивость схемы. Когда это происходит, внешним помехам, таким как электрическое поле, возбуждаемое при запуске двигателя в бытовом приборе, становится легче нарушить внутреннюю работу процессора. Любое снижение напряжения процессора необходимо тщательно контролировать для обеспечения его надежной работы.

Снижение тактовой частоты процессора (в дополнение к любому замедлению тактовой частоты, требуемому из-за снижения напряжения питания) снижает энергопотребление примерно пропорционально. Снижение рабочей частоты также может повысить надежность системы, поскольку для переключения каждого вентиля выделяется больше времени на распространение и достижение конечного состояния, прежде чем оно будет зафиксировано вентилями, которыми он управляет.

Сложное мобильное устройство, такое как смартфон, часто нуждается в быстром переходе между состояниями с высоким и низким энергопотреблением под действием различных входных сигналов, включая действия пользователя, запускаемые по таймеру события и поступление информации по беспроводным соединениям. Для системы выгодно быстро переключаться между режимами высокого и низкого энергопотребления по мере изменения обстоятельств, однако также важно ограничить частоту, с которой происходят такие переходы, поскольку сам акт переключения между этими режимами требует некоторого расхода энергии.

Пользователи ожидают, что их компьютеры и другие сложные цифровые устройства будут потреблять не больше энергии, чем необходимо, но они также рассчитывают на то, что эти системы будут безопасными. В следующем разделе представлены ключевые аспекты управления безопасностью системы.

Управление безопасностью системы

Мы видели, как разделение уровней привилегий между режимами ядра и пользователя поддерживает эффективное разделение приложений, запущенных одним пользователем, от приложений других пользователей и от системных процессов. Это обеспечивает безопасность на уровне выполняемых программ.

В общем и целом это хорошо, но как быть с системами, которые должны оставаться безопасными даже тогда, когда к ним имеют неограниченный физический доступ обычные непроверенные пользователи? Для того чтобы предотвратить доступ любопытных пользователей или злоумышленников к защищенному коду, данным и аппаратным ресурсам, нужно реализовать дополнительные меры на аппаратном уровне.

Прежде чем перейти к подробному описанию средств защиты на аппаратном уровне, полезно перечислить некоторые категории информации и других ресурсов, которые необходимо защищать в цифровых системах.

* **Личная информация.** Такая информация, как государственные идентификационные номера, пароли для доступа к банковским счетам, списки контактов, электронные письма и текстовые сообщения, должна быть защищена даже в случае потери или кражи мобильного устройства, содержащего эту информацию.
* **Деловая информация.** Торговые секреты, списки клиентов, описания экспериментальных продуктов и стратегические планы — вот некоторые категории конфиденциальных бизнес-данных, которые могут иметь большую ценность в руках конкурентов или преступников. Предприятия также собирают большое количество личной информации о своих клиентах и обязаны прилагать значительные усилия для обеспечения безопасности этой информации.
* **Государственная информация.** Государственные организации хранят огромное количество личной информации о своих гражданах и должны гарантировать, что она может быть использована только в разрешенных целях. Правительства государств также накапливают огромные объемы информации, связанной с национальной безопасностью, которая требует применения развернутых протоколов безопасности.

Помимо очевидных мер физической безопасности, связанных с хранением конфиденциальной информации в надежно защищенном помещении с контролем доступа и эффективной системой сигнализации, можно предпринять ряд мер для защиты системы от широкого спектра атак.

Рассмотрим смартфон. Технически грамотный человек может разобрать корпус телефона и получить доступ к аппаратным средствам на уровне электронных схем.

Если этот человек сможет контролировать внешние электрические сигналы процессора и его каналы связи с другими компонентами системы, какие виды информации он сможет собрать? Ответ зависит от типов и количества аппаратных средств защиты, реализованных в конструкции системы.

Первым шагом в проектировании безопасной системы является предотвращение внесения уязвимостей во время разработки. При разработке системы, содержащей встраиваемый процессор, весьма полезно предусмотреть интерфейс **аппаратного отладчика**, который позволяет подключить ПК к устройству с помощью специального интерфейсного кабеля. Используя это соединение, разработчики могут перепрограммировать флеш-память, устанавливать контрольные точки в коде, отображать значения регистров и переменных, а также пошагово просматривать работу кода. Если в выпущенной версии проекта интерфейс для подключения отладчика остается на печатной плате, то пользователи могут подключить к системе собственный отладчик и работать с ним так же, как и разработчики.

Это явно нежелательно для любой системы, предназначенной для безопасной работы. Поскольку возможность подключения отладчика остается весьма полезной да-

же после выпуска продукта, разработчики иногда пытаются оставить сигналы отладчика в схеме, но замаскировать их каким-либо образом, чтобы сделать их функции менее очевидными. В определенной степени этот подход может быть эффективным, однако упорные хакеры продемонстрировали способность обнаруживать наличие даже самых искусно замаскированных отладочных интерфейсов и использовать их для доступа к внутренним устройствам процессора. Наличие доступного интерфейса аппаратной отладки в выпущенном продукте является серьезной уязвимостью безопасности.

Многие производители процессоров начали внедрять средства защиты, чтобы не позволить даже технически грамотным и целенаправленным злоумышленникам нарушить защиту системы. Для обеспечения эффективной защиты разработчики должны внедрять и в полной мере использовать эти возможности в конструкциях своих систем. Вот некоторые примеры технологий безопасности.

- **Защищенный паролем интерфейс аппаратного отладчика.** Некоторые семейства процессоров поддерживают добавление пароля к стандартному интерфейсу аппаратной отладки. Прежде чем такой процессор включит функцию отладки, подключенная система обязана пройти процесс первоначальной аутентификации, в ходе которого она должна предоставить надежный пароль (например, 256-битное число). Это эффективный подход, который сохраняет возможность безопасного поиска и устранения проблем, возникающих после выпуска продукта.
- **Внутренний механизм шифрования с хранилищем ключей.** Некоторые процессоры поддерживают возможность шифрования и дешифрования и хранят секретные ключи для использования во время работы. Эти ключи должны быть сохранены в процессоре во время изготовления системы и после сохранения должны быть недоступны извне.

  Такое сочетание шифровального механизма и хранимых ключей обеспечивает безопасную связь с авторизованными внешними устройствами. Использование высокоскоростного аппаратного шифрования и дешифрования позволяет обеспечить безопасную связь с требуемой скоростью между физически разделенными подсистемами, как, например, в автомобиле.
- **Защита памяти устройства.** Многие семейства процессоров предоставляют несколько вариантов защиты областей памяти. Например, банк ПЗУ, содержащий код, можно заблокировать после программирования, чтобы гарантировать невозможность его последующего перепрограммирования. Области кода также можно заблокировать для чтения в виде данных, но при этом доступ к ним для выполнения может быть сохранен. Процессоры, в которых отсутствует полноценный блок управления памятью, часто оснащают подсистемой, называемой **блоком защиты памяти** (memory protection unit, MPU) и предназначенной для управления требованиями безопасности различных областей памяти процессора.

Несколько возможностей, необходимых для обеспечения безопасной работы системы, были объединены в устройстве под названием Trusted Platform Module (доверенный платформенный модуль), о котором пойдет речь в следующем разделе.

Доверенный платформенный модуль

Современные ПК содержат подсистему, называемую **доверенным платформенным модулем** (Trusted Platform Module, TPM), который реализует различные функции безопасности. TPM — это устойчивый к взлому процессор, разработанный специально для выполнения криптографических операций. В современном ПК он может представлять собой небольшой модуль, подсоединяемый к материнской плате, а иногда он реализован в виде отдельного функционального блока в интегральной схеме главного процессора.

Общие категории функциональных возможностей TPM таковы.

- **Генерирование случайных чисел.** Когда необходимо выбрать большое число, которое будет очень трудно угадать кому-то другому, лучше всего использовать для выбора этого числа действительно случайный метод. Большинство языков программирования предоставляют механизм для генерирования последовательностей чисел, выглядящих как случайные, но в большинстве случаев эти последовательности хотя бы в некоторой степени предсказуемы после получения серии выборок. TPM предоставляет функцию генерирования *истинных* случайных чисел, которая подходит для формирования криптографических ключей, способных обеспечить надежную защиту данных. Генератор истинных случайных чисел использует специализированные аппаратные средства для генерирования непредсказуемой последовательности битов 0 и 1.
- **Генерирование криптографических ключей.** Модуль TPM может создавать криптографические ключи стандартных форматов для использования внутри системы и по запросу пользовательских приложений. Эти ключи обычно применяются для таких целей, как цифровая подпись электронной почты и защита доступа к веб-приложениям.
- **Хранение криптографических ключей.** Помимо генерирования криптографических ключей, TPM может надежно хранить несколько ключей для различных целей. **Шифрование с открытым ключом** использует попарно сгенерированные ключи. Одним из ключей пары является **открытый ключ**, которым можно свободно делиться с кем угодно. Другой ключ, называемый **секретным ключом**, должен быть защищен и предназначен для использования только владельцем ключа. Данные, зашифрованные с помощью открытого ключа, могут быть расшифрованы лишь с помощью секретного ключа. И, аналогично, данные, зашифрованные с помощью секретного ключа, могут быть расшифрованы только с помощью соответствующего открытого ключа. Шифрование с открытым ключом реализует многие возможности обеспечения безопасности, такие как шифрование электронной почты и цифровые

подписи документов. Модуль TPM можно настроить на постоянную блокировку хранимого в нем секретного ключа, что гарантирует невозможность его обнаружения вредоносным программным обеспечением или даже продвинутым злоумышленником, получившим полный физический доступ к системе.

- **Проверка целостности системы.** TPM принимает непосредственное участие в процессе безопасной загрузки. Этот процесс гарантирует, что в ходе загрузки разрешено выполнение только доверенных компонентов встроенного и прочего программного обеспечения. Процесс безопасной загрузки включает в себя вычисление и проверку цифровой подписи каждого используемого во время загрузки компонента встроенного и прочего программного обеспечения до его выполнения.
- **Мониторинг работоспособности системы.** Модуль TPM может гарантировать, что функции безопасности системы, такие как шифрование дисков и безопасная загрузка, будут непрерывно действовать во время работы системы.
- **Услуги аутентификации.** Модуль TPM предоставляет программный интерфейс, предлагающий криптографические функции по мере необходимости для использования приложениями непривилегированного уровня. Этот интерфейс представляет собой безопасный механизм привязки идентификационных данных пользователя к цифровому устройству. Секретный ключ, содержащийся в TPM, обеспечивает надежную, исключающую возможность подделки идентификацию пользователя.

Благодаря TPM современные ПК и другие цифровые устройства можно безопасно использовать в финансовых и конфиденциальных транзакциях по незащищенным каналам связи, включая Интернет.

Главная цель таких механизмов безопасности, как TPM, — пресечь попытки кибератак, целью которых является кража информации или финансовых активов, принадлежащих пользователям компьютерных систем. Спектр возможных атак и некоторые способы защиты от них обсуждаются в следующем разделе.

Противодействие кибератакам

Недостаточно спроектировать компьютерную систему со стандартным набором средств защиты и полагать, что хакеры не смогут преодолеть эти средства защиты. Злоумышленники демонстрируют необычайную сообразительность и используют любые доступные способы, чтобы проникнуть во внутренние механизмы системы, которая кажется им интересной, будь то из интеллектуального любопытства или из преступных побуждений.

Вот некоторые области, которые следует рассмотреть помимо стандартных методов обеспечения безопасности.

- **Анализ энергопотребления, временны́х зависимостей и излучений.** Используя детальный мониторинг энергопотребления системы, времени актив-

ности внешних сигналов или даже радиочастотных излучений, генерируемых во время выполнения алгоритмов, можно провести обратную разработку действий, происходящих внутри процессора. Известны случаи успешных атак, основанных на этих методах, например для извлечения секретных ключей шифрования во время выполнения криптографического алгоритма.

- **Нарушение энергоснабжения.** Атаки, вызывающие колебания или падения напряжения питания во время включения или работы системы, как было показано на практике, приводят к тому, что многие системы оказываются в уязвимом состоянии, в котором некоторые средства защиты становятся неэффективными. Надежная система должна предвидеть такое поведение, естественное или преднамеренное, и возвращаться в безопасное и надежное состояние всякий раз, когда источник питания работает вне пределов установленных параметров.
- **Физическое вмешательство.** Злоумышленник может заменить некоторые компоненты в системе, пытаясь получить повышенный уровень контроля или доступ к важным функциям или данным. Например, замена загрузочного ПЗУ в мобильном устройстве может обеспечить нормальную работу системы и одновременно предоставить злоумышленнику возможность получить неограниченный доступ к системе, используя код в замененном ПЗУ. Все большее число семейств процессоров поддерживают использование цифровых подписей для проверки подлинности кода ПЗУ. Для проверки содержимого ПЗУ во время запуска процессор выполняет криптографический **алгоритм хеширования** над содержимым ПЗУ. Прежде чем будет разрешено выполнение хранящегося в ПЗУ кода, результаты вычисления хеш-функции (сигнатура) должны совпасть с сигнатурой, предварительно загруженной во внутреннюю память процессора. Алгоритм хеширования разработан таким образом, чтобы сделать практически невозможным изменение хранящихся в ПЗУ данных и при этом предоставить ожидаемую сигнатуру. Для дополнительной безопасности содержимое ПЗУ также может быть зашифровано, что не позволит злоумышленнику проанализировать содержащийся в нем код.

В этом разделе были описаны современные подходы, которые широко используются для реализации мер безопасности в цифровых устройствах, и рассмотрены некоторые из более редких уязвимостей, которыми пользуются целеустремленные злоумышленники. Проектирование безопасной системы должно начинаться с твердого усвоения основных элементов безопасности, но также требует большой изобретательности при рассмотрении множества способов, которыми решительно настроенный злоумышленник может попытаться прорвать ее защиту.

Резюме

В этой главе на основании изложенного в предыдущих главах были представлены особенности компьютерной архитектуры, реализующие функциональные требования, характерные для конкретных областей. Мы сосредоточились на расширениях, обычно реализуемых на уровне набора инструкций процессора и во внешних по

отношению к процессору компонентах, которые предоставляют дополнительные возможности помимо общих требований к вычислениям.

Теперь вы должны хорошо ориентироваться в привилегированных режимах процессора и их использовании в многопроцессорных и многопользовательских контекстах, концепциях процессоров и наборов инструкций для вычислений с плавающей запятой, методах управления питанием в устройствах с аккумуляторным питанием, а также в особенностях процессоров и систем, предназначенных для повышения уровня их защиты.

Эти знания подготовили нас к следующей главе, в которой мы рассмотрим наиболее популярные архитектуры процессоров и наборы инструкций, используемые в настоящее время в персональных компьютерах, компьютерах бизнес-класса и интеллектуальных мобильных устройствах. Будут рассмотрены следующие архитектуры: *x*86, *x*64, а также 32- и 64-разрядные варианты ARM.

Упражнения

1. Используя язык программирования, который предоставляет доступ к байтовому представлению типов данных с плавающей запятой (например, C или C++), напишите функцию, которая принимает в качестве входных данных 32-битное значение одинарной точности. Извлеките знак, порядок и мантиссу из байтов этого значения с плавающей запятой и выведите их на устройство отображения. Удалите смещение из порядка перед выводом его значения и выведите мантиссу в виде десятичного числа. Протестируйте программу на значениях `0`, `-0`, `1`, `-1`, `6.674e-11`, `1.0e38`, `1.0e39`, `1.0e-38` и `1.0e-39`. Перечисленные здесь числовые значения, содержащие *e*, используют текстовое представление чисел с плавающей запятой на языке C/C++. Например, `6.674e-11` означает $6{,}674 \times 10^{-11}$.
2. Измените программу из *упражнения 1*, чтобы она могла также принимать значение с плавающей запятой двойной точности и выводить знак, порядок (с удаленным смещением) и мантиссу этого значения. Протестируйте ее с теми же входными значениями, что и в *упражнении 1*, а также со значениями `1.0e308`, `1.0e309`, `1.0e-308` и `1.0e-309`.
3. Найдите в Интернете информацию о семействе процессоров i.MX RT1060 компании NXP Semiconductors. Скачайте технические описания этого семейства и ответьте на следующие вопросы об этих процессорах.
4. Поддерживают ли процессоры i.MX RT1060 концепцию выполнения инструкций в режиме супервизора? Объясните свой ответ.
5. Поддерживают ли процессоры i.MX RT1060 концепцию виртуальной памяти со страничной организацией? Объясните свой ответ.
6. Поддерживают ли процессоры i.MX RT1060 аппаратные вычисления с плавающей запятой? Объясните свой ответ.
7. Какие функции управления питанием поддерживают процессоры i.MX RT1060?
8. Какие функции безопасности поддерживают процессоры i.MX RT1060?

10

Современные архитектуры и наборы инструкций процессоров

Большинство современных персональных компьютеров содержат процессор, поддерживающий 32-разрядную архитектуру *x*86 или 64-разрядную архитектуру *x*64 от Intel или AMD. С другой стороны, почти все смартфоны, умные часы, планшеты и многие встраиваемые системы содержат 32- или 64-разрядные процессоры ARM. В этой главе подробно рассматриваются регистры и наборы инструкций процессоров этих семейств.

Прочитав эту главу, вы получите представление о высокоуровневой организации и уникальных атрибутах регистров, наборов инструкций и языков ассемблера архитектур *x*86, *x*64, 32- и 64-разрядных архитектур ARM, а также о важных аспектах унаследованных функций, поддерживаемых в этих архитектурах.

В этой главе рассматриваются следующие темы:

- архитектура и набор инструкций *x*86;
- архитектура и набор инструкций *x*64;
- архитектура и набор инструкций 32-разрядных процессоров ARM;
- архитектура и набор инструкций 64-разрядных процессоров ARM.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Архитектура и набор инструкций x86

Для целей настоящего обсуждения термин "***x*86**" относится к 16-разрядной и 32-разрядной версиям архитектуры набора инструкций для серии процессоров, которая началась с Intel 8086, представленного в 1978 г. Процессор 8088, выпущенный в 1979 г., функционально очень похож на 8086, за исключением того, что он оснащен 8-разрядной шиной данных вместо 16-разрядной шины в 8086. 8088 был центральным процессором оригинального персонального компьютера IBM PC.

Последующие поколения процессоров этой серии получили названия 80186, 80286, 80386 и 80486, что привело к появлению термина "*x*86" в качестве сокращения для всех членов этого семейства. В дальнейшем было принято решение об отказе от соглашения о цифровых наименованиях, и для следующих поколений использовались названия Pentium, Core, i Series, Celeron и Xeon.

Advanced Micro Devices (**AMD**), компания по производству полупроводников, конкурирующая с Intel, производит *x*86-совместимые процессоры с 1982 г. Некоторые из недавних поколений процессоров AMD *x*86 получили названия Ryzen, Opteron, Athlon, Turion, Phenom и Sempron.

Совместимость в отношении выполнения кода между процессорами Intel и AMD хороша во многих аспектах. Между процессорами этих двух производителей есть некоторые существенные различия, в том числе конфигурация выводов чипов и совместимость с набором микросхем.

Процессоры Intel, как правило, работают только на материнских платах и с наборами микросхем, предназначенными для чипов Intel, а процессоры AMD — только на материнских платах и с наборами микросхем, предназначенными для чипов AMD. Позже в этом разделе мы рассмотрим некоторые другие различия между процессорами Intel и AMD.

8086 и 8088 — это 16-разрядные процессоры, несмотря на 8-разрядную шину данных в 8088. Внутренние регистры в этих процессорах имеют разрядность 16 бит, а набор инструкций работает с 16-битными значениями данных. Процессор 8088 явным образом выполняет два цикла шины для передачи каждого 16-битного значения между процессором и памятью.

8086 и 8088 не поддерживают более сложные функции современных процессоров, такие как виртуальная память со страничной организацией и кольца защиты. Эти ранние процессоры также имели всего 20 адресных строк, что ограничивало объем адресуемой памяти одним мегабайтом. 20-разрядный адрес не может поместиться в 16-разрядный регистр, поэтому для доступа к полному адресному пространству объемом 1 Мбайт необходимо использовать некоторым образом усложненную систему сегментных регистров и смещений.

В 1985 г. компания Intel выпустила процессор 80386 с улучшениями, которые сняли многие из этих ограничений.

В нем были введены следующие возможности.

- **32-разрядная архитектура.** Адреса, регистры и АЛУ имеют разрядность 32 бита, а инструкции изначально работают с операндами разрядностью до 32 бит.
- **Защищенный режим.** Этот режим реализует многоуровневый механизм привилегий, состоящий из колец с номерами от 0 до 3, которые мы рассмотрели в *главе 9*. В Windows и Linux кольцо 0 — это режим ядра, а кольцо 3 — пользовательский режим. Кольца 1 и 2 не используются в этих операционных системах.
- **Блок управления памятью (MMU) на чипе.** В процессоре 80386 блок MMU поддерживает плоскую модель памяти, позволяющую с помощью 32-разрядного адреса получить доступ к любой ячейке в пространстве объемом 4 Гбайт. Манипулирование сегментными регистрами и смещениями больше не требуется. MMU поддерживает виртуальную память со страничной организацией.
- **Трехступенчатый конвейер инструкций.** Этот конвейер ускоряет выполнение инструкций, как описано в *главе 8*.
- **Регистры аппаратной отладки.** Регистры отладки поддерживают настройку до четырех контрольных точек, которые останавливают выполнение кода по указанному виртуальному адресу при обращении к этому адресу и выполнении выбранного условия. Доступные условия остановки — выполнение кода, запись данных и чтение или запись данных. Эти регистры доступны только для использования кодом, запущенным в кольце 0.

Современные процессоры семейства *x*86 при загрузке переводятся в 16-разрядный режим работы оригинального 8086, который теперь называется **реальным режимом**. Он сохраняет совместимость с программным обеспечением, написанным для среды 8086/8088, таким как операционная система MS-DOS.

В большинстве современных систем, работающих на процессорах семейства *x*86, переход в защищенный режим происходит во время запуска системы. После этого перехода операционная система остается в данном режиме до тех пор, пока компьютер не будет выключен.

MS-DOS НА СОВРЕМЕННОМ ПК



Процессор семейства *x*86 в современном ПК совместим на уровне инструкций с оригинальным процессором 8088, однако запуск старой копии MS-DOS в современной компьютерной системе вряд ли будет простой процедурой. Периферийные устройства и их интерфейсы в современных ПК не совместимы с соответствующими интерфейсами в ПК 1980-х годов. MS-DOS потребуется драйвер, который понимает, как взаимодействовать, например, с подключенной через интерфейс USB клавиатурой современной материнской платы.

В наши дни 16-разрядный режим в процессорах *x*86 в основном используется в качестве загрузчика для операционной системы, работающей в защищенном режиме. Поскольку большинство разработчиков компьютерных устройств и программного обеспечения, которое работает на них, вряд ли будут привлечены к реализации такой возможности, оставшаяся часть нашего обсуждения архитектуры *x*86 в этой главе будет посвящена защищенному режиму и связанной с ним плоской модели 32-разрядной памяти.

Архитектура *x*86 поддерживает беззнаковые и знаковые целочисленные типы данных в дополнительном коде разрядностью 8, 16, 32, 64 и 128 бит. Этим типам данных присвоены следующие названия:

- **byte** (байт) — 8 бит;
- **word** (слово) — 16 битов;
- **doubleword** (двойное слово) — 32 бита;
- **quadword** (учетверенное слово) — 64 бита;
- **double quadword** (двойное учетверенное слово) — 128 бит.

В большинстве случаев архитектура *x*86 не предписывает хранение данных этих типов в естественных границах. **Естественная граница** типа данных — это любой адрес, делящийся на размер типа данных в байтах без остатка.

Хранение любого из многобайтовых типов в невыровненных границах разрешено, но не рекомендуется, поскольку это негативно сказывается на производительности: для выполнения инструкций, работающих с невыровненными данными, требуются дополнительные такты. Несколько инструкций, которые работают с двойными учетверенными словами, требуют хранилища с естественным выравниванием и при попытке доступа к невыровненным данным генерируют ошибку общей защиты.

Архитектура *x*86 имеет встроенную поддержку типов данных с плавающей запятой разрядностью 16, 32, 64 и 80 битов. 32-битный, 64-битный и 80-битный форматы были представлены в *главе 9*. 16-битный формат с плавающей запятой называют форматом **половинной точности**, он имеет 11-битную мантиссу, подразумеваемый ведущий бит 1 и 5-битный порядок. Формат с плавающей запятой половинной точности широко используется в обработке данных графическим процессором.

В следующем разделе мы подробно рассмотрим набор регистров архитектуры *x*86.

Набор регистров архитектуры *x*86

В защищенном режиме архитектуры *x*86 предусмотрены восемь 32-разрядных регистров общего назначения, регистр флагов и указатель инструкций. Также предусмотрены шесть сегментных регистров и дополнительные регистры конфигурации, зависящие от конкретной модели процессора. Сегментные регистры и регистры, относящиеся к конкретной модели процессора, настраиваются системным программным обеспечением во время запуска и, как правило, не представляют никакого интереса для разработчиков приложений и драйверов устройств. По этим причи-

нам мы не будем далее обсуждать сегментные регистры и регистры для конкретных моделей процессоров.

16-разрядные регистры общего назначения в оригинальной архитектуре 8086 называются AX, CX, DX, BX, SP, BP, SI и DI. Причина перечисления первых четырех регистров в порядке, не соответствующем алфавитному, заключается в том, что эти восемь регистров помещаются в стек с помощью инструкции pushad (поместить все регистры) именно в таком порядке.

При переходе на 32-разрядную архитектуру процессора 80386 разрядность каждого регистра увеличилась до 32 бит. 32-разрядная версия имени регистра имеет префикс с буквой E для обозначения этого расширения.

Возможен доступ к частям 32-разрядных регистров с меньшей битовой шириной. Например, к младшим 16 битам 32-разрядного регистра EAX можно обращаться по имени AX. Регистр AX, в свою очередь, может быть доступен в виде отдельных байтов, используя имена AH (старший байт) и AL (младший байт). На рис. 10.1 показаны имена регистров и подмножества каждого из них.

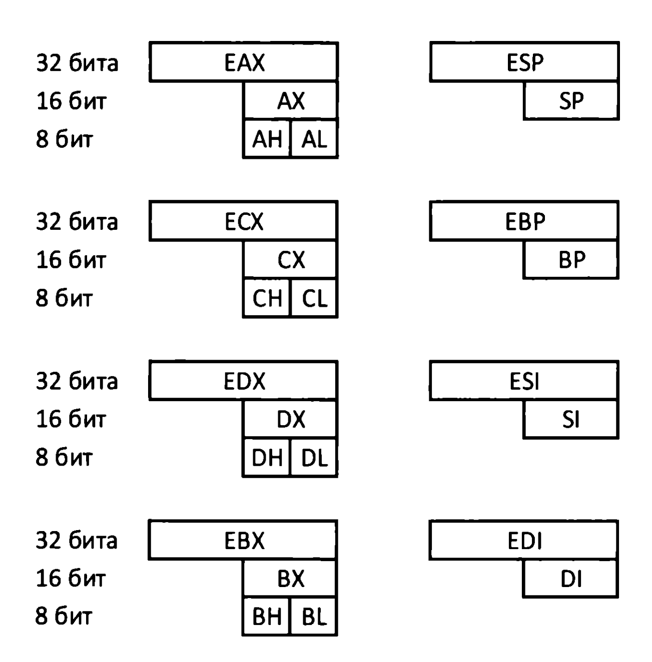


Рис. 10.1. *Имена и подмножества регистров*

Запись в часть 32-разрядного регистра, например в AL, затрагивает только биты в этой части. Загрузка 8-битного значения в AL изменяет младшие 8 бит регистра EAX, оставляя остальные 24 бита без изменений.

В порядке соблюдения традиции архитектуры CISC семейства *x*86 несколько функций, связанных с различными инструкциями, привязаны к определенным регистрам. В табл. 10.1 приведено описание функций, связанных с каждым регистром общего назначения *x*86.

Таблица 10.1. *Регистры общего назначения x86 и связанные с ними функции*

Регистр	Название	Функция
EAX	Аккумулятор	Арифметические операции
ECX	Счетчик	Счетчик циклов и счетчик сдвига/вращения
EDX	Данные	Арифметические операции и операции ввода-вывода
EBX	База	Указатель на данные
ESP	Указатель стека	Указатель на вершину стека
EBP	Указатель базы	Указатель на базу стека внутри функции
ESI	Индекс источника	Указатель на исходное местоположение в операциях с массивами
EDI	Индекс приемника	Указатель на конечное местоположение в операциях с массивами

Эти привязанные к регистрам функции контрастируют с архитектурами многих процессоров RISC, которые, как правило, предлагают большее количество регистров общего назначения. Регистры в RISC-процессоре по большей части функционально эквивалентны друг другу.

Регистр флагов *x*86, EFLAGS, содержит биты состояния процессора, описанные в табл. 10.2.

Таблица 10.2. *Биты регистра флагов x86*

Бит	Название	Функция
0	CF	Флаг переноса: указывает, привело ли сложение к переносу или вычитание к заимствованию. Используется в качестве входных данных инструкциями сложения и вычитания
2	PF	Флаг четности: устанавливается, если младшие 8 бит результата содержат четное число битов со значением 1
4	AF	Флаг вспомогательного переноса: указывает, привело ли сложение к переносу или вычитание к заимствованию из 4 младших битов. Используется в двоично-десятичной арифметике
6	ZF	Флаг нуля: устанавливается, если результат операции равен нулю
7	SF	Флаг знака: устанавливается, если результат операции отрицателен
8	TF	Флаг трассировки: используется при пошаговой отладке
9	IF	Флаг разрешения прерываний: установка этого бита разрешает выполнение аппаратных прерываний

***Таблица 10.2** (окончание)*

Бит	Название	Функция
10	DF	Флаг направления: управляет направлением обработки строк. Если равен 0, используется порядок от низшего адреса к высшему. Если равен 1, используется порядок от самого высшего адреса к низшему
11	OF	Флаг переполнения: устанавливается, если операция привела к переполнению со знаком
12–13	IOPL	Уровень привилегий ввода-вывода: уровень привилегий выполняемого в данный момент потока. Если IOPL имеет значение 0 — это режим ядра, если равен 3 — пользовательский режим
14	NT	Флаг вложенной задачи: управляет цепочкой прерываний
16	RF	Флаг возобновления: используется для обработки исключений во время отладки
17	VM	Флаг виртуального режима 8086: если установлен, включен режим совместимости с процессором 8086. Этот режим позволяет запускать некоторые приложения MS-DOS в контексте операционной системы с защищенным режимом
18	AC	Флаг контроля выравнивания: если установлен, включена функция контроля выравнивания данных в памяти. Например, если флаг AC установлен, сохранение 16-битного значения по нечетному адресу вызывает исключение контроля выравнивания. Когда этот флаг не установлен, процессоры *x*86 могут выполнять доступ к памяти без выравнивания, но количество тактов, требуемых для выполнения инструкций, при этом может увеличиться
19	VIF	Флаг разрешения виртуальных прерываний: виртуальная версия флага IF в виртуальном режиме 8086
20	VIP	Флаг ожидающего виртуального прерывания: устанавливается, когда прерывание находится в состоянии ожидания в виртуальном режиме 8086
21	ID	Флаг идентификации: если этот бит может быть установлен, поддерживается инструкция cpuid. Инструкция cpuid возвращает идентификатор процессора и информацию о функциях

Все биты в регистре EFLAGS, которые отсутствуют в табл. 10.2, зарезервированы и не используются.

32-битный указатель инструкций EIP содержит адрес следующей инструкции для выполнения, если не было выбрано ответвление. Когда выполняется условие ответвления, в EIP загружается адрес назначения ветви и выполнение продолжается с этого адреса.

В архитектуре *x*86 принят прямой порядок байтов. это означает, что многобайтовые значения хранятся в памяти с наименьшим значащим байтом по низшему адресу и наиболее значимым байтом по высшему адресу.

Режимы адресации x86

Как и следовало ожидать от архитектуры CISC, *x*86 поддерживает несколько режимов адресации. Существует несколько правил, связанных с адресацией операндов-источников и операндов-приемников, которым необходимо следовать для получения правильных инструкций. Например, размеры операнда-источника и операнда-приемника в инструкции mov должны быть равны. Ассемблер попытается выбрать подходящий размер для операнда, который имеет неопределенный размер (например, непосредственное значение 7), чтобы он соответствовал разрядности целевого местоположения (например, 32-битного регистра EAX). В случаях, когда размер операнда не может быть определен, для указания размера необходимо использовать ключевые слова, такие как byteptr.

Язык ассемблера в этих примерах использует *синтаксис Intel*, в котором принято размещение операндов в порядке "приемник — источник". Синтаксис Intel используется в основном в контекстах Windows и MS-DOS. В альтернативной нотации, известной как *синтаксис AT&T*, операнды размещаются в порядке "источник — приемник". Этот синтаксис используется в операционных системах на основе UNIX. Во всех примерах в этой книге используется синтаксис Intel.

Архитектура *x*86 поддерживает несколько режимов адресации, которые мы рассмотрим далее. Комментарии в коде ассемблера начинаются с точки с запятой и продолжаются до конца строки.

Неявная адресация

В этом режиме адресации регистр определяется опкодом инструкции. Пример:

```
clc ; обнуление флага переноса (CF в регистре флагов EFLAGS)
```

Регистровая адресация

Один или оба регистра, источник и приемник, закодированы в инструкции:

```
mov eax, ecx ; копирование содержимого регистра ECX в регистр EAX
```

Регистры можно использовать в качестве первого операнда, второго операнда или обоих операндов.

Непосредственная адресация

В качестве операнда инструкции подставляется непосредственное значение:

```
mov eax, 7 ; перемещение 32-битного значения 7 в регистр EAX
mov ax, 7 ; перемещение 16-битного значения 7 в регистр AX
```

При использовании синтаксиса Intel не требуется ставить перед непосредственными значениями символ #.

Адресация с прямым доступом к памяти

В качестве операнда инструкции подставляется адрес значения:

```
mov eax, [078bch] ; копирование 32-битного значения по шестнадцатеричному адресу
                  ; 78BC в регистр EAX
```

В ассемблере *x*86 заключение выражения в квадратные скобки означает, что выражение является адресом. При выполнении перемещений или других операций над операндами, заключенными в квадратные скобки, значение, над которым производится операция, — это данные по указанному адресу. Исключением из этого правила является инструкция LEA (загрузка действительного адреса), которую мы рассмотрим позже.

Косвенная регистровая адресация

Операндом является регистр, содержащий адрес значения данных:

```
mov eax, [esi]    ; копирование 32-битного значения по адресу, указанному
                  ; в регистре ESI, в регистр EAX
```

Этот режим эквивалентен использованию указателя для ссылки на переменную в C или C++.

Индексная адресация

Операнд указывает регистр и смещение, сочетание которых определяет адрес значения данных:

```
mov eax, [esi + 0bh]    ; копирование 32-битного значения по адресу (ESI + 0bh)
                        ; в регистр EAX
```

Этот режим удобен для доступа к элементам структуры данных. В этом сценарии регистр ESI содержит адрес структуры, а добавленная константа — байтовое смещение элемента от начала этой структуры.

Относительная индексная адресация

Операнд указывает базовый регистр, индексный регистр и смещение, которые суммируются для определения адреса значения данных:

```
mov eax, [ebx + esi + 10]    ; копирование 32-битного значения с началом
                             ; по адресу (EBX + ESI + 10) в регистр EAX
```

Режим удобен для доступа к отдельным элементам данных в массиве структур данных. В этом примере регистр EBX содержит адрес начала массива структур, регистр ESI содержит смещение нужной структуры в массиве, а постоянное значение (10) представляет собой смещение нужного элемента данных от начала выбранной структуры.

Относительная индексная адресация с масштабированием

Операнд состоит из базового регистра, индексного регистра, умноженного на масштабный коэффициент, и смещения, которые суммируются для определения адреса значения данных:

```
mov eax, [ebx + esi*4 + 10]    ; копирование 32-битного значения с началом
                               ; по адресу (EBX + ESI*4 + 10) в регистр EAX
```

В этом режиме адресации перед суммированием с другими компонентами адреса операнда значение в индексном регистре может быть умножено на 1 (по умолчанию), 2, 4 или 8. Использование масштабного коэффициента не приводит к снижению производительности. Эта возможность удобна при итерации по массивам, содержащим элементы размером 2, 4 или 8 байт.

В режимах относительной адресации в качестве базового или индексного регистра можно использовать большинство регистров общего назначения.

На рис. 10.2 показаны возможные комбинации использования регистров и масштабирования в режимах относительной адресации.

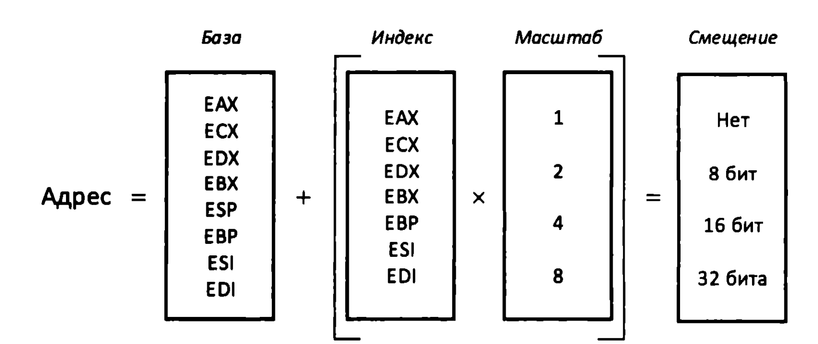


Рис. 10.2. Режим относительной адресации

Все восемь регистров общего назначения можно использовать в качестве базового регистра. И из этих восьми только ESP нельзя применять в качестве индексного регистра.

Категории инструкций x86

Набор инструкций *x*86 был представлен вместе с процессором Intel 8086 и за прошедшие годы несколько раз расширялся. Некоторые из наиболее значительных изменений связаны с расширением архитектуры с 16 до 32 бит, при котором также были добавлены защищенный режим и виртуальная память со страничной организацией. Почти во всех случаях новые возможности добавлялись с сохранением полной обратной совместимости.

Полный набор инструкций *x*86 содержит несколько сотен инструкций. Мы не будем обсуждать их все в этой главе. В этом разделе приведены краткие обзоры наиболее важных и часто встречающихся инструкций, применимых к приложениям, работающим в пользовательском режиме, и драйверам устройств.

Это подмножество инструкций *x*86 можно разделить на несколько общих категорий: перемещение данных, манипулирование стеком, арифметика и логика, преобразования, поток управления, манипулирование строками и флагами, ввод-вывод, защищенный режим. Мы также рассмотрим несколько вспомогательных инструкций, которые не относятся к какой-либо определенной категории.

Перемещение данных

Инструкции перемещения данных не влияют на флаги процессора. Перемещение данных выполняют следующие инструкции.

- mov — копирование значения данных, на которое указывает второй операнд, в ячейку памяти, указанную в качестве первого операнда.
- cmov*cc* — условное перемещение данных второго операнда в регистр, указанный в качестве первого операнда, если условие *cc* истинно. Условие определяется по одному или нескольким из следующих флагов процессора: CF, ZF, SF, OF и PF. Используются следующие коды условий: e (равно), ne (не равно), g (больше), ge (больше или равно), a (выше), ae (выше или равно), l (меньше), le (меньше или равно), b (ниже), be (ниже или равно), o (переполнение), no (без переполнения), z (ноль), nz (не ноль), s (SF = 1), ns (SF = 0), cxz (регистр CX содержит ноль) и ecxz (регистр ECX содержит ноль).
- movsx, movzx — это варианты инструкции mov, выполняющие расширение знака и дополнение нулями, соответственно. Операнд-источник должен быть меньшего размера, чем операнд-приемник.
- lea — вычисление адреса, предоставляемого вторым операндом, и сохранение его в ячейку памяти, указанную в первом операнде. Второй операнд заключен в квадратные скобки. В отличие от других инструкций перемещения данных, по адресу назначения хранится вычисленный адрес, а не значение данных.

Манипулирование стеком

Инструкции манипулирования стеком не влияют на флаги процессора. К этой категории относятся следующие инструкции:

* push — уменьшение содержимого регистра ESP на 4, а затем помещение 32-битного операнда в ячейку стека, на которую указывает ESP.
* pop — копирование 32-битного значения данных, на которое указывает содержимое регистра ESP в местоположение операнда (регистр или адрес в памяти), а затем увеличение содержимого ESP на 4.
* pushfd, popfd — помещение в стек или извлечение из стека регистра EFLAGS.
* pushad, popad — помещение в стек или извлечение из стека регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI в указанном порядке.

Арифметика и логика

Арифметические и логические инструкции меняют флаги процессора. Следующие инструкции выполняют арифметические и логические операции.

* add, sub — целочисленное сложение и вычитание. При вычитании второй операнд вычитается из первого. Оба операнда могут быть регистрами, или один операнд может быть ячейкой памяти, а другой — регистром. Один операнд может быть константой.
* adc, sbb — целочисленное сложение и вычитание с использованием флага CF в качестве входных данных — признака переноса (для сложения) или заимствования (для вычитания).
* cmp — вычитание одного операнда из другого с отбрасыванием результата и одновременным обновлением флагов OF, SF, ZF, AF, PF и CF на основании результата.
* neg — отрицание операнда.
* inc, dec — увеличение или уменьшение операнда на единицу.
* mul — умножение беззнаковых целых чисел. Размер произведения зависит от размера операнда. Операнд размером в байт умножается на AL, а результат помещается в AX. Операнд размером в слово умножается на AX, а результат помещается в DX:AX со старшими 16 битами в DX. Операнд размером в двойное слово умножается на EAX, а результат помещается в EDX:EAX.
* imul — умножение знаковых целых чисел. Первый операнд должен быть регистром, и в него же помещается результат операции. Всего может быть два или три операнда. В форме с двумя операндами первый операнд умножается на второй операнд, а результат сохраняется в первом операнде (регистре). В форме с тремя операндами второй операнд умножается на третий операнд, а результат сохраняется в первом операнде (регистре). В форме с тремя операндами третий операнд должен быть непосредственным значением.

- div, idiv — беззнаковое (div) или знаковое (idiv) деление. Размер результата зависит от размера операнда. Операнд размером в байт делится на AX, частное помещается в AL, а остаток — в AH. Операнд размером в слово делится на DX:AX, частное помещается в AX, а остаток — в DX. Операнд размером в двойное слово делится на EDX:EAX, частное помещается в EAX, а остаток — в EDX.
- and, or, xor — соответствующая логическая операция над двумя операндами и сохранение результата в местоположении операнда-приемника.
- not — логическая операция НЕ (инверсия битов) над одним операндом.
- sal, shl, sar, shr — логический (shl и shr) или арифметический (sal и sar) сдвиг аргумента размером в байт, слово или двойное слово влево или вправо на заданное число позиций — от 1 до 31 бита. sal и shl помещают последний сдвинутый бит во флаг переноса и вставляют нули в освободившиеся младшие значащие биты. shr помещает последний сдвинутый бит во флаг переноса и вставляет нули в освободившиеся старшие биты. sar отличается от shr распространением бита знака в освободившиеся старшие биты.
- rol, rcl, ror, rcr — операция вращения влево или вправо на 0–31 бит, по выбору — с помощью флага переноса. rcl и rcr выполняют операцию вращения с использованием флага переноса, а rol и ror — без его использования.
- bts, btr, btc — считывание бита с заданным номером (представленным в качестве второго операнда) из первого операнда во флаг переноса, а затем установка (bts), обнуление (btr) или дополнение (btc) этого бита. Этим инструкциям может предшествовать ключевое слово lock, делающее данную операцию неделимой.
- test — логическая операция И над двумя операндами и обновление состояния флагов SF, ZF и PF в зависимости от результата.

Преобразования

Инструкции преобразования увеличивают размерность значений данных. К этой категории относятся следующие инструкции.

- cbw — преобразование байта (регистр AL) в слово (регистр AX).
- cwd — преобразование слова (регистр AX) в двойное слово (регистр DX).
- cwde — преобразование слова (регистр AX) в двойное слово (регистр EAX).
- cdq — преобразование двойного слова (регистр AX) в учетверенное слово (регистр EDX:EAX).

Поток управления

Инструкции потока управления осуществляют условную или безусловную передачу выполнения на определенный адрес.

- jmp — передача управления инструкции, расположенной по адресу, который указан в качестве операнда.

* jcc — передача управления инструкции, расположенной по адресу, который указан в качестве операнда, если условие *cc* истинно. Коды условий были приведены ранее в описании инструкции cmov*cc*. Условие определяется по одному или нескольким из следующих флагов процессора: CF, ZF, SF, OF и PF.
* call — помещение текущего значения регистра EIP в стек и передача управления инструкции по адресу, указанному в качестве операнда.
* ret — извлечение верхнего значения из стека и сохранение его в регистре EIP. Если имеется операнд, эта инструкция извлекает указанное количество байтов из стека для очистки параметров.
* loop — уменьшение на единицу значения счетчика loop в регистре ECX и, если оно не равно нулю, передача управления инструкции, расположенной по адресу, который указан в качестве операнда.

Манипулирование строками

Инструкции манипулирования строками могут иметь префикс в виде ключевого слова rep для повторения операции заданное число раз, которое указывается в регистре ECX, с увеличением или уменьшением на единицу местоположения источника и приемника на каждой итерации, в зависимости от состояния флага DF. Размер операнда, обрабатываемого на каждой итерации, может быть равен байту, слову или двойному слову. Исходный адрес каждого строкового элемента задается в регистре ESI, а адрес назначения — в регистре EDI. К этой категории относятся следующие инструкции:

* mov — перемещение строкового элемента.
* cmps — сравнение элементов в соответствующих позициях в пределах двух строк.
* scas — сравнение строкового элемента со значением в регистре EAX, AX или AL, в зависимости от размера операнда.
* lods — загрузка строки в регистр EAX, AX или AL, в зависимости от размера операнда.
* stos — сохранение содержимого регистра EAX, AX или AL, в зависимости от размера операнда, по адресу, записанному в регистр EDI.

Манипулирование флагами

Инструкции манипулирования флагами изменяют биты в регистре EFLAGS.

* stc, clc, cmc — установка, обнуление или дополнение флага переноса CF.
* std, cld — установка или обнуление флага направления DF.
* sti, cli — установка или обнуление флага прерывания IF.

Ввод-вывод

Инструкции ввода-вывода считывают данные из периферийных устройств или записывают в них данные.

- `in`, `out` — перемещение 1, 2 или 4 байтов между регистром `EAX`, `AX` или `AL` и портом ввода-вывода в зависимости от размера операнда.
- `ins`, `outs` — перемещение элемента данных между памятью и портом ввода-вывода таким же образом, как это делают инструкции для работы со строками.
- `rep ins`, `rep outs` — перемещение блоков данных между памятью и портом ввода-вывода таким же образом, как это делают инструкции для работы со строками.

Защищенный режим

Следующие инструкции открывают доступ к функциям защищенного режима.

- `sysenter`, `sysexit` — передача управления с кольца 3 на кольцо 0 (`sysenter`) или с кольца 0 на кольцо 3 (`sysexit`) в процессорах Intel.
- `syscall`, `sysret` — передача управления с кольца 3 на кольцо 0 (`syscall`) или с кольца 0 на кольцо 3 (`sysret`) в процессорах AMD. В режиме *x*86 (32-разрядном) процессоры AMD также поддерживают инструкции `sysenter` и `sysexit`.

Вспомогательные инструкции

Эти инструкции не подпадают под категории, перечисленные выше.

- `int` — инициирование программного прерывания. Операнд — это номер вектора прерывания.
- `nop` — без операции.
- `cpuid` — предоставление информации о модели процессора и его возможностях.

Другие категории инструкций

Инструкции, перечисленные в этом разделе, являются одними из наиболее распространенных в приложениях *x*86 и драйверах устройств, помимо описанных в предыдущих разделах. Архитектура *x*86 содержит широкое разнообразие категорий инструкций, включая следующие:

- **инструкции вычислений с плавающей запятой** — эти инструкции выполняются в блоках вычислений с плавающей запятой *x*87;
- **инструкции SIMD** — к этой категории относятся инструкции MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2 и AVX-512; некоторые из наборов инструкций в этой категории были представлены в *разд. "Модель обработки SIMD" главы 8*;

- **инструкции AES** — эти инструкции поддерживают шифрование и дешифрование с использованием **усовершенствованного стандарта шифрования** (advanced encryption standard, AES);
- **инструкции MPX**: **расширения для защиты памяти** (memory protection eXtensions, MPX) улучшают целостность памяти, предотвращая ошибки, такие как переполнение буфера;
- **инструкции SMX**: **расширения безопасного режима** (Safer mode eXtensions, SMX) повышают безопасность системы при наличии решений, связанных с доверием пользователям;
- **инструкции TSX**: **расширения для синхронизации транзакций** (transactional synchronization eXtensions, TSX) повышают производительность при многопоточном выполнении с использованием общих ресурсов;
- **инструкции VMX**: **расширения для виртуальных машин** (virtual machine eXtensions, VMX) обеспечивают безопасную и эффективную поддержку виртуализированных операционных систем.

Для инструкций с плавающей запятой и SIMD предусмотрены дополнительные регистры процессора.

Помимо перечисленных здесь существует еще ряд категорий инструкций *x*86, часть которых была удалена в более поздних поколениях архитектуры.

Общие шаблоны использования инструкций

Ниже приведены примеры шаблонов использования инструкций, с которыми вы будете часто сталкиваться в скомпилированном коде. Методы, показанные в этих примерах, дают желаемый результат, сводя при этом к минимуму размер кода и требуемое количество тактов:

```
xor reg, reg  ; обнуление reg
test reg, reg ; проверка reg на равенство нулю
add reg, reg  ; сдвиг reg влево на один бит
```

Форматы инструкций x86

Отдельные инструкции *x*86 имеют переменную длину, которая может меняться от 1 до 15 байт. Компоненты одной инструкции, включая любые необязательные байты, размещаются в памяти в перечисленной далее последовательности.

- **Байты префикса.** Один или несколько необязательных байтов префикса предоставляют вспомогательную информацию о выполнении операции с данным опкодом. Например, префикс `lock` блокирует шину в многопроцессорной системе для выполнения неделимых операций типа "проверить — установить". Префикс `rep` и его варианты позволяют строковым инструкциям выполнять повторяющиеся операции над строковыми элементами в рамках

одной инструкции. Другие префиксы предоставляют указания для инструкций условного ветвления или для переопределения принятого по умолчанию размера адреса или операнда.

- **Байты кода операции (опкода).** За любыми байтами префикса следует опкод архитектуры *x*86, состоящий из 1–3 байтов. Для некоторых кодов операций дополнительные три бита опкода кодируются в байте *ModR/M*, следующим за этим опкодом.
- **Байт ModR/M.** Этот байт требуется не для всех инструкций. Байт *ModR/M* содержит три информационных поля, предоставляющих информацию о режиме адресации и регистровых операндах. Два старших бита этого байта (поле *Mod*) и три его младших бита (поле *R/M*) объединяются, формируя поле из 5 бит с 32 возможными значениями. Из них 8 значений идентифицируют регистровые операнды, а остальные 24 значения определяют режимы адресации. Оставшиеся 3 бита (поле *reg/opcode*) указывают регистр либо содержат три дополнительных бита опкода в зависимости от инструкции.
- **Байты смещения адреса.** 0, 1, 2 или 4 байта обеспечивают смещение адреса, используемое при вычислении адреса операнда.
- **Байты непосредственного значения.** Если инструкция включает в себя непосредственное значение, оно находится в последних 1, 2 или 4 байтах инструкции.

Переменная длина инструкций *x*86 делает процесс их декодирования довольно сложным. Кроме того, у инструментов отладки могут возникнуть затруднения с разбором последовательности инструкций в обратном порядке, что может потребоваться для отображения кода, предшествующего контрольной точке.

Эти трудности возникают из-за того, что завершающая последовательность байтов в длинных инструкциях может сформировать полную допустимую инструкцию. Такие сложности являются заметным отличием от более привычных форматов инструкций, используемых в архитектурах RISC.

Язык ассемблера x86

На языке ассемблера можно разрабатывать программы любого уровня сложности.

Однако большинство современных приложений в значительной степени или полностью разрабатывают на языках высокого уровня. Язык ассемблера, как правило, используют в тех случаях, когда возникает необходимость в специализированных инструкциях или требуется экстремальная оптимизация, недостижимая с помощью оптимизирующего компилятора.

Независимо от языка, используемого при разработке приложений, весь код в конечном счете должен выполняться в виде инструкций процессора. Для полного понимания того, как выполняется код в компьютерной системе, нет иного пути, кроме изучения состояния системы после выполнения каждой отдельной инструкции. Хо-

рошим способом научиться понимать и работать в этой среде является написание некоторого кода на ассемблере.

Пример на языке ассемблера *x*86 в следующем листинге представляет собой законченное приложение *x*86, которое запускается из командной консоли Windows, печатает строку текста и затем завершает работу.

```
.386
.model FLAT,C
.stack 400h

.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near extern exit:near

public main
main proc
    ; Печать сообщения
    push   offset message
    call   printf

    ; Выход из программы с кодом состояния 0
    push   0
    call   exit
main endp

.data
message db "Привет, архитектор компьютеров!",0

end
```

Ниже приведено описание содержимого этого файла на языке ассемблера.

- Директива `.386` указывает, что инструкции в этом файле следует интерпретировать как инструкции для процессоров 80386 и более поздних поколений.
- Директива `.model FLAT,C` определяет 32-разрядную плоскую модель памяти и использование соглашений о вызове функций на языке C.
- Директива `.stack 400h` определяет размер стека в 400h (1024) байт.
- Директива `.code` указывает на начало исполняемого кода.

- Директивы includelib и extern ссылаются на предоставляемые системой библиотеки и функции в них, которые будут использоваться программой.
- Директива public указывает, что имя функции main является видимым извне.
- Строки между main proc и main endp содержат инструкции на языке ассемблера, составляющие функцию main.
- Директива .data указывает на начало памяти данных. Выражение message db определяет строку сообщения как последовательность байтов, за которой следует нулевой байт.
- Директива end отмечает окончание программы.

Этот файл с именем hello_x86.asm транслируется и компонуется для получения исполняемого файла программы hello_x86.exe с помощью следующей команды, которая запускает Microsoft Macro Assembler:

```
ml /Fl /Zi /Zd hello_x86.asm
```

Компоненты этой команды:

- ml запускает ассемблер (ml.exe);
- /Fl создает файл листинга;
- /Zi включает в исполняемый файл отладочную информацию об используемых в программе именах;
- /Zd включает в исполняемый файл отладочную информацию о номерах строк;
- hello_x86.asm — это имя исходного файла на языке ассемблера.

Ниже показана часть файла листинга hello_x86.lst, сгенерированного ассемблером.

```
                            .386
                            .model FLAT,C
                            .stack 400h
 00000000                   .code
                            includelib libcmt.lib
                            includelib legacy_stdio_definitions.lib

                            extern printf:near
                            extern exit:near

                            public main
 00000000                   main proc
                                ; Печать сообщения
 00000000  68 00000000 R        push   offset message
 00000005  E8 00000000 E        call   printf
```

```
                          ; Выход из программы с кодом состояния 0
0000000A  6A 00            push   0
0000000C  E8 00000000 E    call   exit
00000011                  main endp

00000000                .data
00000000 48 65 6C 6C 6F message db "Привет, архитектор компьютеров!",0
         2C 20 43 6F 6D
         70 75 74 65 72
         20 41 72 63 68
         69 74 65 63 74
         21 00
```

В левом столбце этого листинга отображаются смещения адресов от начала функции `main`. В строках, содержащих инструкции, за смещением адреса следует код операции. Адресные ссылки в коде (например, `offset message`) отображаются в листинге в виде `00000000`, т. к. их значения определяются во время компоновки, а не в ходе трансляции, когда создается этот листинг.

Ниже показаны выходные данные, отображаемые при запуске программы.

```
C:\>hello_x86.exe
Привет, архитектор компьютеров!
```

Далее мы рассмотрим расширение 32-разрядной архитектуры *x*86 до 64-разрядной архитектуры *x*64.

Архитектура и набор инструкций x64

Исходная спецификация для процессорной архитектуры, расширяющая процессор *x*86 и набор инструкций до 64 разрядов и получившая название **AMD64**, была представлена компанией AMD в 2000 г. Первый процессор AMD64, Opteron, был выпущен в 2003 г. Intel, обнаружившая свое отставание от AMD, разработала архитектуру, совместимую с AMD64, которая в итоге была названа **Intel 64**. Первым процессором Intel с 64-разрядной архитектурой стал Xeon, представленный в 2004 г. Архитектурам AMD и Intel дали общее название ***x*86-64**, отражающее развитие от *x*86 до 64 бит. В общепринятом использовании этот термин был сокращен до ***x*64**.

Первая версия Linux, поддерживающая архитектуру *x*64, была выпущена в 2001 г., задолго до появления первых процессоров *x*64. В Windows поддержка архитектуры *x*64 была добавлена в 2005 г.

Процессоры, реализующие архитектуры AMD64 и Intel 64, в значительной степени совместимы на уровне набора инструкций для программ пользовательского режи-

ма. Существует несколько различий между этими архитектурами, наиболее значительным из которых является разница в поддержке инструкций Intel sysenter/sysexit и инструкций AMD syscall/sysret, которую мы обсуждали ранее.

В целом операционные системы и компиляторы языков программирования справляются с этими различиями, поэтому они редко становятся предметом внимания разработчиков программного обеспечения и систем. Разработчики программного обеспечения ядра, драйверов и ассемблерного кода должны учитывать эти различия.

Основные особенности архитектуры *x*64 таковы.

- *x*64 — это в основном совместимое 64-разрядное расширение 32-разрядной архитектуры *x*86. Большинство программ, особенно приложений пользовательского режима, написанных для 32-разрядной среды, должны выполняться без изменений на процессоре, работающем в 64-разрядном режиме. 64-разрядный режим также называют **длинным режимом**.
- Восемь 32-разрядных регистров общего назначения *x*86 в *x*64 расширены до 64 разрядов. На 64-разрядные регистры указывает префикс имени регистра R. Например, в *x*64 расширенный регистр *x*86 EAX называется RAX. Подкомпоненты регистров *x*86 EAX, AX, AH и AL остались доступными и в *x*64.
- Указатель инструкций RIP расширен до 64 бит. Регистр флагов RFLAGS также охватывает 64 бита, хотя его старшие 32 бита зарезервированы. Младшие 32 бита регистра RFLAGS аналогичны EFLAGS в архитектуре *x*86.
- Были добавлены восемь 64-разрядных регистров общего назначения, получивших названия от R8 до R15.
- 64-разрядные целые числа поддерживаются в качестве собственного типа данных.
- Процессоры *x*64 сохраняют возможность работы в режиме совместимости с *x*86. Этот режим поддерживает 32-разрядные операционные системы и позволяет любому приложению, созданному для *x*86, работать на процессорах *x*64. В режиме совместимости с 32-разрядной средой 64-разрядные расширения недоступны.

Виртуальные адреса в архитектуре *x*64 имеют разрядность 64 бита, поддерживая адресное пространство размером 16 **эксабайт (Эбайт)**, что эквивалентно 2^{64} байтам. Однако современные процессоры AMD и Intel поддерживают только 48 бит виртуального адресного пространства. Это ограничение снижает аппаратную сложность процессора, сохраняя при этом поддержку до 256 **терабайт (Тбайт)** виртуального адресного пространства. Процессоры текущего поколения также поддерживают максимум 48 бит физического адресного пространства. Это позволяет процессору обрабатывать 256 Тбайт физической оперативной памяти, хотя современные материнские платы не поддерживают количество устройств DRAM, которое может потребоваться такой системе.

Набор регистров архитектуры x64

В архитектуре *x*64 расширение размера регистров *x*86 до 64 бит и добавление регистров R8–R15 дают карту регистров, показанную на рис. 10.3.

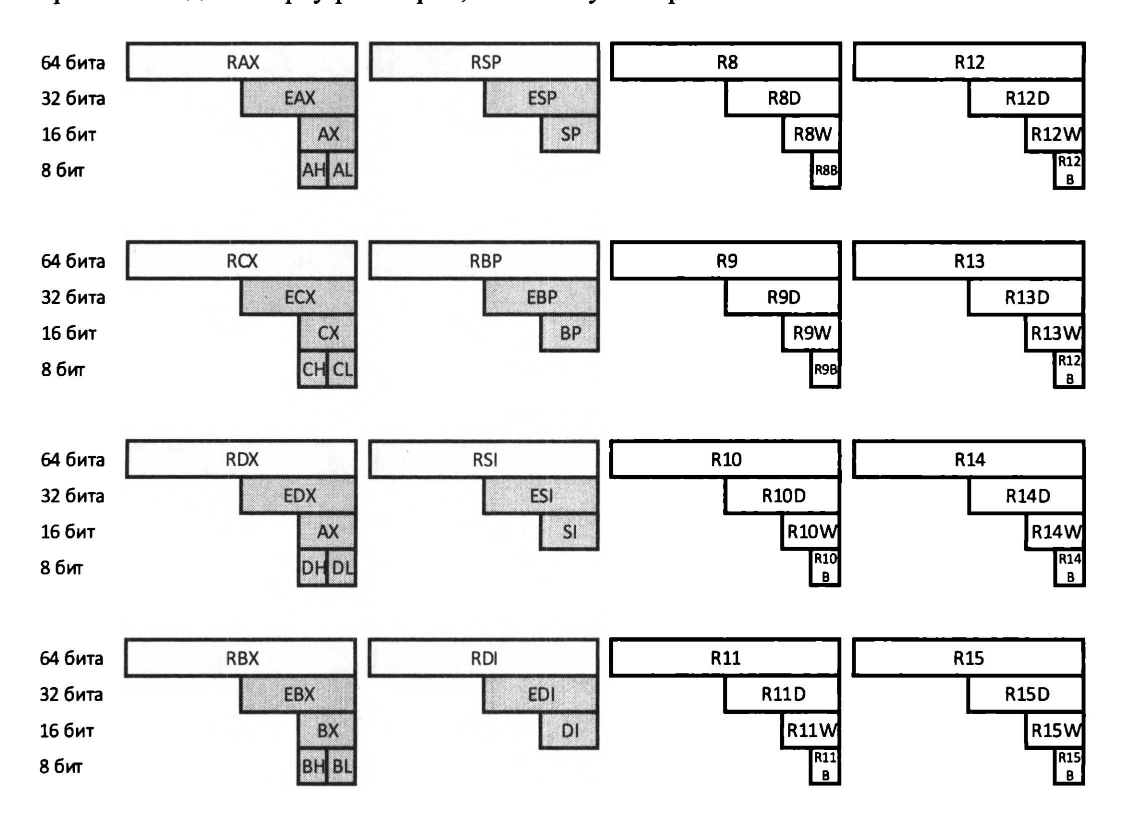


Рис. 10.3. *Регистры архитектуры x64*

На рис. 10.3 регистры *x*86, описанные в предыдущем разделе (и присутствующие в *x*64), имеют более темный фон. При работе в 64-разрядном режиме регистры *x*86 имеют прежние имена и размеры.

Расширенные 64-разрядные версии регистров *x*86 имеют имена, начинающиеся с буквы R. Предусмотрена возможность доступа к меньшим частям новых регистров 64-разрядной архитектуры (R8–R15) с помощью соответствующей буквы суффикса:

- суффикс **D** обеспечивает доступ к младшим 32 битам регистра: R11D;
- суффикс **W** обеспечивает доступ к младшим 16 битам регистра: R11W;
- суффикс **B** обеспечивает доступ к младшим 8 битам регистра: R11B.

В отличие от регистров *x*86, новые регистры в архитектуре *x*64 имеют действительно общее назначение и не выполняют никаких специальных функций на уровне инструкций процессора.

Категории и форматы инструкций x64

Архитектура *x*64 реализует по существу тот же набор инструкций, что и *x*86, но с расширениями для 64-разрядной среды. При работе в 64-разрядном режиме архитектура *x*64 по умолчанию использует адреса размером 64 бита и операнды размером 32 бита. Новый байт префикса `rex` кода операции определяет использование 64-разрядных операндов.

Формат инструкций *x*64 в памяти соответствует формату архитектуры *x*86 с некоторыми незначительными для наших целей исключениями. Добавление поддержки байта префикса `rex` является наиболее значительным отличием от формата инструкций *x*86. Смещения адресов и непосредственные значения в некоторых инструкциях могут иметь разрядность 64 бита в дополнение ко всем прочим вариантам разрядности, поддерживаемым в *x*86.

Несмотря на возможность определить инструкции длиной более 15 байт, декодер инструкций процессора выдаст ошибку общей защиты, если будет предпринята попытка декодировать инструкцию длиной более 15 байт.

Язык ассемблера x64

Исходный файл на ассемблере *x*64 для программы `hello` похож на версию этого кода для *x*86, с некоторыми примечательными отличиями.

- Отсутствует директива, определяющая модель памяти, поскольку в *x*64 существует только одна модель памяти.
- В **интерфейсе программирования приложений** (application programming interface, API) Windows *x*64 используется соглашение о вызове, которое сохраняет первые четыре аргумента вызываемой функции в регистрах `RCX`, `RDX`, `R8` и `R9` в указанном порядке. Этот подход отличается от принятого по умолчанию соглашения о вызовах *x*86, согласно которому параметры помещаются в стек. Обе библиотечные функции, вызываемые этой программой (`printf` и `exit`), получают один аргумент, передаваемый через регистр `RCX`.
- Соглашение о вызове требует, чтобы вызывающая функцию программа выделяла пространство стека для хранения как минимум того количества аргументов, которые передаются вызываемой функции, с минимальным резервированием пространства под четыре аргумента, даже если число передаваемых аргументов меньше. Поскольку стек растет в памяти вниз, для этого требуется вычитание из указателя стека. Такое резервирование стека выполняет инструкция `sub rsp, 40`. После возврата из вызванной функции для отмены этого резервирования обычно требуется корректировка указателя стека. Наша программа для завершения выполнения вызывает функцию `exit`, что делает этот шаг ненужным.

Код 64-разрядной версии программы hello выглядит следующим образом:

```
.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near
extern exit:near

public main
main proc
    ; Резервирование места в стеке
    sub    rsp, 40

    ; Печать сообщения
    lea    rcx, message
    call   printf

    ; Выход из программы с кодом состояния 0
    xor    rcx, rcx
    call   exit
main endp

.data
message db "Привет, архитектор компьютеров!",0

end
```

Этот файл с именем `hello_x64.asm` транслируется и компонуется для получения исполняемого файла программы `hello_x64.exe` с помощью следующего вызова Microsoft Macro Assembler (версия *x*64):

```
ml64 /Fl /Zi /Zd hello_x64.asm
```

Компоненты этой команды:

- `ml64` запускает 64-разрядный ассемблер;
- `/Fl` создает файл листинга;
- `/Zi` включает в исполняемый файл отладочную информацию об используемых в программе именах;
- `/Zd` включает в исполняемый файл отладочную информацию о номерах строк;
- `hello_x64.asm` — это имя исходного файла на языке ассемблера.

Ниже показана часть файла листинга `hello_x64.lst`, сгенерированного этой командой ассемблера:

```
00000000                .code
                        includelib libcmt.lib
                        includelib legacy_stdio_definitions.lib

                        extern printf:near
                        extern exit:near

                        public main
00000000                main proc
                            ; Резервирование места в стеке
00000000  48/ 83 EC 28      sub    rsp, 40

                            ; Печать сообщения
00000004  48/ 8D 0D         lea    rcx, message
          00000000 R
0000000B  E8 00000000 E     call   printf

                             ; Выход из программы с кодом состояния 0
00000010  48/ 33 C9          xor    rcx, rcx
00000013  E8 00000000 E      call   exit
00000018                main endp

00000000               .data
00000000 48 65 6C 6C 6F message db "Привет, архитектор компьютеров!",0
         2C 20 43 6F 6D
         70 75 74 65 72
         20 41 72 63 68
         69 74 65 63 74
         21 00
```

Результат запуска этой программы выглядит следующим образом:

```
C:\>hello_x64.exe
Привет, архитектор компьютеров!
```

На этом наше краткое знакомство с архитектурами *x*86 и *x*64 завершается. Но здесь еще есть чему поучиться — издание *"Intel 64 and IA-32 Architectures Software Developer's Manual", volumes 1–4 ("Руководство разработчика программного обес-*

печения для архитектур Intel 64 и IA-32", тома 1–4) содержит почти 5000 страниц подробной документации по этим архитектурам. В этой главе мы затронули лишь малую часть этой темы.

Далее мы проведем аналогичную обзорную экскурсию по 32-разрядным и 64-разрядным архитектурам ARM.

Архитектура и набор инструкций 32-разрядных процессоров ARM

Архитектуры ARM определяют семейство RISC-процессоров, подходящих для решения самых разнообразных практических задач. Процессоры, основанные на архитектуре ARM, предпочтительны в решениях, где требуется сочетание высокой производительности, низкого энергопотребления и компактных физических размеров.

Архитектуры ARM разработала британская компания по производству полупроводников и программного обеспечения ARM Holdings, которая предоставляет лицензии на свою продукцию другим компаниям, изготавливающим процессоры. Архитектура ARM находит широкое применение в виде **систем на кристалле** (System-On-Chip, SoC), сочетающих процессор со специализированными аппаратными средствами для реализации таких функций, как сотовая радиосвязь в смартфонах.

Процессоры ARM используются в самых разных областях — от крошечных устройств на батарейках до суперкомпьютеров. Эти процессоры работают в качестве встроенных процессоров в критически важных для безопасности системах, таких как антиблокировочные тормозные системы автомобилей, и в качестве процессоров общего назначения в умных часах, мобильных телефонах, планшетах, ноутбуках, настольных компьютерах и серверах. По состоянию на 2021 г. было произведено более 180 млрд процессоров ARM.

Процессоры ARM — это полноценные RISC-системы с большим набором регистров общего назначения и однотактным выполнением большинства инструкций. Стандартные инструкции ARM имеют фиксированный размер 32 бита, хотя для систем, где приоритет отдается объему памяти, доступен отдельный набор инструкций переменной длины **T32** (ранее его называли **Thumb**). Набор инструкций T32 сочетает в себе 16- и 32-разрядные инструкции.

Процессоры ARM текущего поколения поддерживают как набор инструкций ARM, так и T32 и могут переключаться между этими двумя наборами в процессе работы. Большинство операционных систем и приложений предпочитают использовать набор инструкций T32, а не набор ARM, поскольку это помогает повысить плотность кода.

ARM — это **архитектура загрузки/сохранения**, требующая загрузки данных из памяти в регистр, прежде чем с ними могут быть выполнены какие-либо операции, например обработка в АЛУ. Последующая инструкция сохраняет результат обратно в память. Это может показаться шагом назад по сравнению с архитектурами *x*86

и *x*64, которые работают непосредственно с операндами в памяти в рамках одной инструкции, однако на практике подход с загрузкой/сохранением позволяет с высокой скоростью выполнять несколько последовательных операций над операндом сразу после его загрузки в один из многих регистров процессора.

Процессоры ARM поддерживают переключаемый порядок байтов. Предусмотрен параметр конфигурации для выбора прямого или обратного порядка байтов для многобайтовых значений. Значение по умолчанию — прямой порядок байтов, т. к. эту конфигурацию обычно используют операционные системы.

Архитектура ARM имеет встроенную поддержку следующих типов данных:

- **byte** (байт) — 8 бит;
- **halfword** (полуслово) — 16 бит;
- **word** (слово) — 32 бита;
- **doubleword** (двойное слово) — 64 бита.



КАКОВА ДЛИНА СЛОВА?

Между типами данных в архитектуре ARM и архитектурах *x*86 и *x*64 существует разница, которая может вызвать путаницу: в *x*86 и *x*64 слово имеет длину 16 бит, а двойное слово — 32 бита. В ARM слово состоит из 32 бит, а двойное слово — из 64 бит.

Процессоры ARM поддерживают восемь режимов работы с различными привилегиями выполнения. Ниже приведен список этих режимов и их сокращения:

- **пользовательский режим** (user, USR);
- **режим супервизора** (supervisor, SVC);
- **режим быстрого прерывания** (fast interrupt request, FIQ);
- **режим обычного прерывания** (interrupt request, IRQ);
- **режим мониторинга** (monitor, MON);
- **режим останова** (abort, ABT);
- **неопределенный режим** (UND);
- **системный режим** (system, SYS).

Для целей операционных систем и пользовательских приложений наиболее важными уровнями привилегий являются USR и SVC. Два режима запроса прерываний, FIQ и IRQ, используются драйверами устройств для обработки прерываний.

В большинстве операционных систем, работающих на процессорах ARM, включая Windows и Linux, ядро функционирует в режиме ARM SVC, эквивалентном кольцу 0 в *x*86/*x*64. Режим USR в ARM эквивалентен кольцу 3 в *x*86/*x*64. Приложения, работающие под управлением Linux на процессорах ARM, используют программные

прерывания для запроса сервисов ядра, что предполагает переход из режима USR в режим SVC.

Архитектура ARM предоставляет системные возможности, выходящие за рамки возможностей главного процессора, посредством концепции сопроцессоров. Каждый сопроцессор реализует специализированную категорию функциональных возможностей, обеспечивая поддержку главного процессора. В системе может быть реализовано до 16 сопроцессоров, четырем из которых назначены предопределенные функции.

Сопроцессор 15 реализует блок управления памятью (MMU) и другие системные функции. При наличии в системе сопроцессор 15 должен поддерживать опкоды инструкций, набор регистров и варианты поведения, установленные для MMU. Сопроцессоры 10 и 11 объединяются для реализации операций с плавающей запятой в процессорах, оснащенных этой функцией. Сопроцессор 14 обеспечивает поддержку функций отладки.

За прошедшие годы было выпущено несколько версий архитектуры ARM. В настоящее время широко используется версия архитектуры **ARMv8-A**, поддерживающая 32- и 64-разрядные операционные системы и приложения. 32-разрядные приложения могут работать под управлением 64-разрядной операционной системы ARMv8-A.

Практически все высококлассные смартфоны и мобильные устройства, выпускавшиеся с 2016 г., спроектированы на базе процессоров или систем на кристалле (SOC), основанных на архитектуре ARMv8-A. В следующем повествовании основное внимание будет уделено 32-разрядному режиму ARMv8-A. Отличия 64-разрядного режима ARMv8-A мы рассмотрим в одном из следующих разделов этой главы.

Набор регистров ARM

В режиме USR архитектура ARM имеет шестнадцать 32-разрядных регистров общего назначения: от `R0` до `R15`. Первые 13 регистров имеют действительно общее назначение, а остальные три выполняют следующие функции.

- `R13` — это указатель стека, также называемый `SP` в ассемблерном коде. Этот регистр указывает на вершину стека.
- `R14` — это регистр связи, также называемый `LR`. Он содержит адрес возврата во время выполнения вызываемой функции. Использование регистра связи отличается от архитектуры *x*86/*x*64, где адрес возврата помещается в стек.

  Причина использования регистра для хранения адреса возврата заключается в том, что после завершения функции значительно быстрее возобновить выполнение по адресу в `LR`, чем извлечь адрес возврата из стека и возобновить выполнение по этому адресу.
- `R15` — это программный счетчик, также называемый `PC`. Из-за конвейерной обработки значение, содержащееся в `PC`, обычно на две инструкции опережает выполняемую в данный момент инструкцию. В отличие от *x*86/*x*64, поль-

зовательский код может напрямую считывать и записывать данные в регистр PC. Запись адреса в PC приводит к немедленному переходу выполнения к только что записанному адресу.

Регистр текущего состояния программы (current program status register, CPSR) содержит биты состояния и управления режимом, аналогичные EFLAGS/RFLAGS в архитектурах *x*86/*x*64 (табл. 10.3).

Таблица 10.3. *Отдельные биты регистра CPSR*

Бит	Название	Функция
0–3	M	Режим: текущий уровень привилегий выполнения (USR, SVC и т. д.)
4	T	"Thumb": установлен, если активен набор инструкций T32 (Thumb). Если этот бит обнулен, то активен набор инструкций ARM. Установить и обнулить этот бит можно с помощью пользовательского кода
9	E	Порядок байтов: установка этого бита включает режим с обратным порядком байтов. Если этот бит обнулен, то включен режим с прямым порядком байтов. В большинстве программ используется режим с прямым порядком байтов
27	Q	Флаг кумулятивного насыщения: устанавливается, если в какой-то момент в серии операций произошло переполнение или насыщение
28	V	Флаг переполнения: устанавливается, если операция привела к переполнению со знаком
29	C	Флаг переноса: указывает, привело ли сложение к переносу или вычитание к заимствованию
30	Z	Флаг нуля: устанавливается, если результат операции равен нулю
31	N	Флаг отрицательного знака: устанавливается, если результат операции отрицателен

Биты регистра CPSR, не указанные в табл. 10.3 либо зарезервированы, либо относятся к функциям, не обсуждаемым в этой главе.

По умолчанию большинство инструкций не влияют на эти флаги. Для того чтобы результат выполнения инструкции оказал влияние на флаги, к ней необходимо добавить суффикс s. Например, в случае инструкции сложения: adds. Инструкции сравнения являются исключением из этого правила; они обновляют флаги автоматически.

Режимы адресации ARM

В стиле истинной концепции RISC единственные инструкции ARM, которые могут получить доступ к системной памяти, — это инструкции загрузки данных в регистры и сохранения данных из них.

Инструкция `ldr` загружает в регистр данные из памяти, а инструкция `str` — сохраняет содержимое регистра в ячейку памяти. Отдельная инструкция, `mov`, перемещает данные из одного регистра в другой или помещает в регистр непосредственное значение.

При вычислении целевого адреса для операции загрузки или сохранения ARM начинает с базового адреса, указанного в регистре, и добавляет приращение для получения целевого адреса памяти. Существует три метода определения приращения, которое должно быть добавлено к содержимому базового регистра в инструкциях загрузки и сохранения данных регистра.

- **Смещение.** В базовый регистр добавляется знаковая константа. Смещение сохраняется как часть инструкции. Например, инструкция `ldr r0, [r1, #10]` загружает в `r0` слово по адресу `r1+10`. Как показано в следующих примерах режима адресации, целевой адрес в базовом регистре может быть проиндексирован до или после обращения к ячейке памяти.
- **Регистр.** Беззнаковое приращение, хранящееся в регистре, может быть добавлено к значению в базовом регистре или вычтено из него. Например, инструкция `ldr r0, [r1, r2]` загружает в `r0` слово по адресу `r1+r2`. В качестве базового регистра можно рассматривать любой из регистров.
- **Регистр с масштабированием.** Перед добавлением к значению в базовом регистре или вычитанием из него приращение в регистре сдвигается влево или вправо на указанное количество битовых позиций. Например, инструкция `ldr r0, [r1, r2, lsl #3]` загружает в `r0` слово по адресу `r1+(r2×8)`. Можно использовать логический сдвиг влево или вправо, `lsl` или `lsr`, с дополнением освободившихся битовых позиций нулями, или арифметический сдвиг вправо, `asr`, с повторением знакового бита в освободившихся позициях.

В следующих разделах представлены режимы адресации, используемые для указания операндов-источников и операндов-приемников в инструкциях ARM.

Непосредственное значение

Непосредственное значение вводится как часть инструкции. Возможные непосредственные значения представляют собой закодированные в инструкции 8-битные значения с вращением на четное число битовых позиций. Полное 32-разрядное значение указать нельзя, т. к. сама инструкция имеет ширину не более 32 бит. Для того чтобы загрузить в регистр произвольное 32-разрядное значение, следует использовать инструкцию `ldr` для загрузки значения из памяти:

```
mov r0, #10 // Загрузка в r0 десятичного значения 10 в 32-разрядном формате
mov r0, #0xFF000000 // Загрузка в r0 32-разрядного значения FF000000h
```

Второй пример содержит 8-битное значение `FFh` в опкоде инструкции. Во время выполнения осуществляется поворот влево на 24 битовые позиции до старших 8 бит этого слова.

Прямая регистровая адресация

В этом режиме один регистр копируется в другой:

```
mov r0, r1 // Копирование r1 в r0
mvn r0, r1 // Копирование НЕ(r1) в r0
```

Косвенная регистровая адресация

В регистре предоставляется адрес операнда. Обозначение регистра, содержащего адрес, заключено в квадратные скобки:

```
ldr r0, [r1] // Загрузка в r0 32-разрядного значения по адресу, указанному в r1
str r0, [r3] // Сохранение r0 по адресу, указанному в r3
```

В отличие от большинства инструкций, str использует первый операнд в качестве источника, а второй — в качестве приемника.

Косвенная регистровая адресация со смещением

Адрес операнда вычисляется путем добавления смещения к базовому регистру:

```
ldr r0, [r1, #32] // Загрузка в r0 значения по адресу [r1+32]
str r0, [r1, #4]  // Сохранение r0 по адресу [r1+4]
```

Косвенная регистровая адресация со смещением и преинкрементом

Адрес значения определяется путем добавления смещения к базовому регистру. В базовый регистр помещается вычисленный адрес, и этот адрес используется для загрузки значения в целевой регистр:

```
ldr r0, [r1, #32]! // Загрузка в r0 [r1+32] и обновление r1 до (r1+32)
str r0, [r1, #4]!  // Сохранение r0 по адресу [r1+4] и обновление r1 до (r1+4)
```

Косвенная регистровая адресация со смещением и постинкрементом

Сначала базовый адрес используется для доступа к ячейке памяти. Затем в базовый регистр помещается вычисленный адрес:

```
ldr r0, [r1], #32 // Загрузка [r1] в r0, затем обновление r1 до (r1+32)
str r0, [r1], #4  // Сохранение r0 по адресу [r1], затем обновление r1 до (r1+4)
```

Косвенная регистровая адресация с двумя регистрами

Адрес операнда представляет собой сумму базового регистра и регистра приращения. Имена регистров заключены в квадратные скобки.

```
ldr r0, [r1, r2] // Загрузка в r0 адреса [r1+r2]
str r0, [r1, r2] // Сохранение r0 по адресу [r1+r2]
```

Косвенная регистровая адресация с двумя регистрами и масштабированием

Адрес операнда представляет собой сумму базового регистра и регистра приращения со сдвигом влево или вправо на заданное число битов. Имена регистров и информация о сдвиге заключены в квадратные скобки.

```
ldr r0, [r1, r2, lsl #5] // Загрузка в r0 адреса [r1+(r2*32)]
str r0, [r1, r2, lsr #2] // Сохранение r0 по адресу [r1+(r2/4)]
```

В следующем разделе представлены общие категории инструкций ARM.

Категории инструкций ARM

Инструкции, описанные в этом разделе, взяты из набора T32.

Загрузка/сохранение

Эти инструкции перемещают данные между регистрами и памятью.

- `ldr`, `str` — перемещение 8-битного (суффикс `b` для байта), 16-битного (суффикс `h` для полуслова) или 32-битного значения между регистром и ячейкой памяти. `ldr` копирует значение из памяти в регистр, а `str` копирует содержимое регистра в память. `ldrb` копирует 1 байт в младшие 8 бит регистра.
- `ldm`, `stm` — загрузка или сохранение содержимого нескольких регистров. С их помощью можно копировать от 1 до 16 регистров в память или из нее. Например, инструкция `ldm r1, {r0, r2, r4-r11}` загружает в регистры `r0`, `r2` и от `r4` до `r11` данные из непрерывной области памяти, начиная с адреса, указанного в `r1`. С помощью этих инструкций можно загрузить из памяти или сохранить в ней содержимое любого подмножества регистров.

Манипулирование стеком

Эти инструкции сохраняют данные в стеке и извлекают их из него.

- `push`, `pop` — помещение в стек или извлечение из стека любого подмножества регистров, например: `push {r0, r2, r4-r11}`. Эти инструкции являются вариантами инструкций `ldm` и `stm`.

Перемещение между регистрами

Эти инструкции перемещают данные между регистрами.

- mov, mvn — перемещение содержимого регистра (mov) или результата побитовой инверсии содержимого регистра (mvn) в регистр назначения.

Арифметика и логика

Эти инструкции, как правило, имеют один регистр назначения и два операнда-источника. Первый операнд-источник — это регистр, а второй операнд может быть регистром, сдвинутым регистром или непосредственным значением.

При добавлении суффикса s эти инструкции устанавливают флаги условий. Например, adds выполняет сложение и устанавливает флаги условий.

- add, sub — сложение или вычитание двух чисел. Например, инструкция `add r0, r1, r2, lsl #3` эквивалентна выражению $r_0 = r_1 + (r_2 \times 2^3)$. Оператор lsl выполняет логический сдвиг второго операнда, r2, влево.
- adc, sbc — сложение или вычитание двух чисел с переносом или заимствованием.
- neg — отрицание числа.
- and, orr, eor — логические операции И, ИЛИ и "исключающее ИЛИ".
- orn, eon — логические операции ИЛИ и "исключающее ИЛИ" с использованием первого операнда и результата побитовой инверсии второго операнда.
- bic — обнуление выбранных битов в регистре.
- mul — перемножение двух чисел.
- Mla — перемножение двух чисел с накоплением результата. Эта инструкция содержит дополнительный операнд для указания регистра-накопителя.
- sdiv, udiv — знаковое и беззнаковое деление соответственно.

Сравнение

Эти инструкции сравнивают два значения и устанавливают флаги условий на основе результата сравнения. Суффикс s в этих инструкциях для установки флагов условий не требуется.

- cmp — вычитание одного числа из другого с отбрасыванием результата и установкой флагов условий. Эта инструкция эквивалентна инструкции subs за исключением отбрасывания результата.
- cmn — сложение двух чисел с отбрасыванием результата и установкой флагов условий. Эта инструкция эквивалентна инструкции adds, за исключением отбрасывания результата.
- tst — побитовая операция И с отбрасыванием результата и установкой флагов условий. Эта инструкция эквивалентна инструкции ands, за исключением отбрасывания результата.

Поток управления

Эти инструкции выполняют условную или безусловную передачу управления по целевому адресу.

- b — безусловный переход к целевому адресу.
- b*cc* — условный переход на основании одного из следующих кодов условий, обозначенных *cc*: eq (равно), ne (не равно), gt (больше), lt (меньше), ge (больше или равно), le (меньше или равно), cs (установка флага переноса), cc (обнуление флага переноса), mi (минус: флаг N = 1), pl (плюс: флаг N = 0), vs (флаг V установлен), vc (флаг V обнулен), hi (выше: флаг C установлен, а флаг Z обнулен) или ls (ниже или равно: флаг C обнулен, и флаг Z обнулен).
- bl — переход по указанному адресу с сохранением адреса следующей инструкции в регистре связи (r14, также называемом lr). Для возврата из вызываемой функции в вызывающий код используется инструкция mov pc, lr.
- bx — переход с выбором набора инструкций. Если бит 0 целевого адреса равен 1, выполняется переключение в режим T32. Если бит 0 равен 0, выполняется переключение в режим ARM. Бит 0 адресов инструкций всегда должен быть равен нулю из-за требований ARM к выравниванию адресов. Это освобождает бит 0 для выбора набора инструкций.
- blx — переход по ссылке с выбором набора инструкций. Эта инструкция сочетает в себе функции инструкций bl и bx.

Режим супервизора

С помощью этой инструкции код пользовательского режима может инициировать вызов режима супервизора.

- svc (**вызов режима супервизора**) — инициирование программного прерывания, которое заставляет обработчик исключений режима супервизора обрабатывать запрос на обслуживание системы.

Контрольная точка

Эта инструкция используется отладчиками при разработке программного обеспечения.

- bkpt (**вызов контрольной точки**). Эта инструкция использует 16-битный операнд, с помощью которого отладочное программное обеспечение может идентифицировать контрольную точку.

Условное выполнение

Многие инструкции ARM поддерживают условное выполнение с использованием тех же кодов условий, что и в инструкциях ветвления. Если условие инструкции имеет значение "ложь", инструкция обрабатывается как пустая операция. Код условия добавляется к обозначению инструкции. Этот механизм условного выполнения формально известен как **предикация**.

Например, эта функция преобразует **полубайт** (младшие 4 бита байта) в соответствующий его значению ASCII-символ:

```
// Преобразование младших 4 битов r0 в символ ascii в r0
nibble2ascii:
and r0, #0xF
cmp r0, #10
addpl r0, r0, #('A' - 10)
addmi r0, r0, #'0'
mov pc, lr
```

Инструкция `cmp` вычитает `10` из полубайта в `r0` и устанавливает флаг `N`, если `r0` меньше 10. Если `r0` больше или равно 10, флаг `N` обнуляется.

Если флаг `N` равен нулю, выполняется инструкция `addpl` (`pl` означает "плюс", т. е. "неотрицательный"), а инструкция `addmi` не выполняется. Если флаг `N` установлен, инструкция `addpl` не выполняется, а инструкция `addmi` выполняется. После завершения этой последовательности регистр `r0` содержит символ в диапазоне 0–9 или A–F.

Использование условного выполнения инструкций помогает поддерживать эффективную работу конвейера инструкций, избегая ветвления.

Другие категории инструкций

Процессоры ARM в качестве опции поддерживают ряд инструкций SIMD и инструкций для операций с плавающей запятой. Поддерживаются также дополнительные инструкции, которые обычно используют только во время настройки системы.

32-разрядный язык ассемблера ARM

Следующий пример на ассемблере ARM использует синтаксис GNU Assembler, поставляемого вместе с **интегрированной средой разработки** (integrated development environment, IDE) Android Studio. Другие ассемблеры могут использовать иной синтаксис. Как и в случае с синтаксисом языков ассемблера Intel для *x*86 и *x*64, порядок операндов для большинства инструкций — сначала приемник, затем источник.

Ниже приведен текст файла исходной программы `hello` на языке ассемблера ARM:

```
.text
.global _start

_start:
    mov     r0, #1       // int fd 1 (stdout)
    ldr     r1, =message // const void *buf
    mov     r2, #count   // size_t count
    mov     r7, #4       // syscall 4 (sys_write)
```

```
    svc     0
    mov     r0, #0        // int status (0=OK)
    mov     r7, #1        // syscall 1 (sys_exit)
    svc     0
.data
message:
    .ascii   "Привет, архитектор компьютеров!"
count = . - message
```

Этот файл с именем `hello_arm.s` транслируется и компонуется для получения исполняемого файла программы `hello_arm` с помощью следующих команд. Эти команды используют средства разработки, поставляемые в пакете **Android Studio Native Development Kit** (**NDK**). Предполагается, что переменная среды Windows `PATH` включает в себя каталог со средствами разработки NDK.

```
arm-linux-androideabi-as -al=hello_arm.lst -o hello_arm.o hello_arm.s
arm-linux-androideabi-ld -o hello_arm hello_arm.o
```

Назначение компонентов этих команд:

- `arm-linux-androideabi-as` — запускает ассемблер;
- `-al=hello_arm.lst` — создает файл листинга с именем `hello_arm.lst`;
- `-o hello_arm.o` — создает объектный файл с именем `hello_arm.o`;
- `hello_arm.s` — это имя исходного файла на языке ассемблера;
- `arm-linux-androideabi-ld` — запускает компоновщик;
- `-o hello_arm` — создает объектный файл с именем `hello_arm`;
- `hello_arm.o` — это имя объектного файла, предоставляемого компоновщику в качестве входных данных.

Ниже показана часть файла листинга `hello_arm.lst`, сгенерированного этой командой ассемблера:

```
 1                  .text
 2                  .global _start
 3
 4                  _start:
 5 0000 0100A0E3      mov     r0, #1        // int fd 1 (stdout)
 6 0004 14109FE5      ldr     r1, =message // const void *buf
 7 0008 1A20A0E3      mov     r2, #count    // size_t count
 8 000c 0470A0E3      mov     r7, #4        // syscall 4 (sys_write)
 9 0010 000000EF      svc     0
10
11 0014 0000A0E3      mov     r0, #0        // int status (0=OK)
```

```
12 0018 0170A0E3     mov     r7, #1       // syscall 1 (sys_exit)
13 001c 000000EF     svc     0
14
15              .data
16              message:
17 0000 48656C6C    .ascii   "Привет, архитектор компьютеров!"
17      6F2C2043
17      6F6D7075
17      74657220
17      41726368
18              count = . - message
```

Эту программу можно запустить на устройстве Android с включенным **режимом разработчика**. Здесь мы не будем вдаваться в процедуру включения этого режима, но вы сможете получить необходимую информацию по этой теме с помощью поиска в Интернете.

Следующие строки отображаются при запуске этой программы на устройстве Android с процессором ARM, подключенном к ПК с помощью USB-кабеля:

```
C:\>adb push hello_arm /data/local/tmp/hello_arm
C:\>adb shell chmod +x /data/local/tmp/hello_arm
C:\>adb shell /data/local/tmp/hello_arm
Привет, архитектор компьютеров!
```

Эти команды используют инструмент **Android Debug Bridge** (**adb**), входящий в пакет Android Studio. Программа `hello_arm` запускается на устройстве Android, однако выходные данные программы передаются на ПК и отображаются в окне командной строки.

В следующем разделе представлена 64-разрядная архитектура ARM, являющаяся расширением 32-разрядной архитектуры ARM.

Архитектура и набор инструкций 64-разрядных процессоров ARM

О выходе 64-разрядной версии архитектуры ARM, названной **AArch64**, было объявлено в 2011 г. Она имеет тридцать один 64-разрядный регистр общего назначения, 64-разрядную адресацию, 48-разрядное виртуальное адресное пространство и новый набор инструкций, названный **A64**.

Набор 64-разрядных инструкций является надмножеством набора 32-разрядных инструкций, благодаря чему существующий 32-разрядный код может выполняться на 64-разрядных процессорах без изменений.

Инструкции имеют ширину 32 бита, а большинство операндов — 32 или 64 бита. Функции регистров A64 в некоторых отношениях отличаются от 32-разрядного режима: программный счетчик больше не доступен напрямую в виде регистра, также предусмотрен дополнительный регистр, который всегда возвращает нулевое значение операнда.

На уровне привилегий пользователя большинство инструкций A64 имеют те же обозначения, что и соответствующие 32-разрядные инструкции. Ассемблер определяет, с какими данными работает инструкция — 64-разрядными или 32-разрядными — на основе предоставленных операндов. Длину операнда и размер регистра, используемые инструкцией, определяют следующие правила:

- имена 64-разрядных регистров начинаются с буквы `x`; например, `x0`;
- имена 32-разрядных регистров начинаются с буквы `w`; например, `w1`;
- 32-разрядные регистры занимают младшие 32 бита 64-разрядного регистра с соответствующим номером.

При работе с 32-разрядными регистрами применяются следующие правила.

- Операции с регистрами, такие как сдвиг вправо, выполняются так же, как и в 32-разрядной архитектуре. 32-разрядный арифметический сдвиг вправо использует в качестве знакового бита бит 31, а не бит 63.
- Коды условий для 32-разрядных операций устанавливаются на основе результата в младших 32 битах.
- При записи в регистр W старшие 32 бита соответствующего регистра X заполняются нулями.

A64 — это архитектура загрузки/сохранения с теми же названиями инструкций для операций с памятью (`ldr` и `str`), что и для 32-разрядного режима. Существуют некоторые различия и ограничения по сравнению с 32-разрядными инструкциями загрузки и хранения.

- Базовый регистр должен быть регистром X (64-разрядным).
- Смещение адреса может быть задано теми же способами, что и в 32-разрядном режиме, а также в регистре X. 32-разрядное смещение может быть дополнено до 64 битов нулями или знаковыми битами.
- В режимах индексной адресации в качестве смещения можно использовать только непосредственные значения.
- В инструкциях `ldm` и `stm` A64 не поддерживает загрузку и сохранение нескольких регистров в одной инструкции. Вместо этого в A64 добавлены инструкции `ldp` и `stp` для загрузки и сохранения нескольких регистров в одной инструкции.
- A64 поддерживает условное выполнение только для небольшого подмножества инструкций.

Операции со стеком в A64 существенно отличаются. Возможно, самое большое отличие в этой области заключается в том, что при доступе к данным указатель стека должен сохранять 16-байтовое выравнивание.

64-разрядный язык ассемблера ARM

Ниже приведен текст файла исходной программы hello на 64-разрядном языке ассемблера ARM:

```
.text
.global _start

_start:
    // Печать сообщения в файл 1 (stdout) с помощью системного вызова 64
    mov     x0, #1
    ldr     x1, =msg
    mov     x2, #msg_len
    mov     x8, #64
    svc     0

    // Выход из программы с помощью системного вызова 93 с возвращением состояния 0
    mov     x0, #0
    svc     0

.data
msg:
    .ascii      "Привет, архитектор компьютеров!"
msg_len = . - msg
```

Этот файл с именем hello_arm64.s транслируется и компонуется для получения исполняемого файла программы hello_arm64 с помощью следующих команд. Эти команды используют 64-разрядные средства разработки, поставляемые в пакете Android Studio NDK. Их использование предполагает, что переменная среды Windows PATH включает в себя каталог с необходимыми средствами.

```
aarch64-linux-android-as -al=hello_arm64.lst -o hello_arm64.o ^ hello_arm64.s
aarch64-linux-android-ld -o hello_arm64 hello_arm64.o
```

Назначение компонентов этих команд:

- aarch64-linux-android-as — запускает ассемблер;
- -al=hello_arm64.lst — создает файл листинга с именем hello_arm64.lst;
- -o hello_arm64.o — создает объектный файл с именем hello_arm64.o;
- hello_arm64.s — это имя исходного файла на языке ассемблера;
- aarch64-linux-android-ld — запускает компоновщик;
- -o hello_arm64 — создает исполняемый файл с именем hello_arm64;

* hello_arm64.o — это имя объектного файла, предоставляемого компоновщику в качестве входных данных.

Ниже показана часть файла листинга hello_arm64.lst, сгенерированного ассемблером.

```
 1                .text
 2                .global _start
 3
 4                _start:
 5                    // Печать сообщения в файл 1 (stdout) с помощью системного вызова 64
 6 0000 200080D2          mov    x0, #1
 7 0004 E1000058          ldr    x1, =msg
 8 0008 420380D2          mov    x2, #msg_len
 9 000c 080880D2          mov    x8, #64
10 0010 010000D4          svc    0
11
12                    // Выход из программы с помощью системного вызова 93 с возвращением состояния 0
13 0014 000080D2          mov    x0, #0
14 0018 A80B80D2          mov    x8, #93
15 001c 010000D4          svc    0
16
17                .data
18                msg:
19 0000 48656C6C          .ascii     "Привет, архитектор компьютеров!"
19      6F2C2043
19      6F6D7075
19      74657220
19      41726368
20                msg_len = . - msg
```

Эту программу можно запустить на устройстве Android с включенным **режимом разработчика**, как было описано ранее. Следующие строки отображаются при запуске этой программы на устройстве Android с процессором ARM, подключенном к ПК с помощью USB-кабеля:

```
C:\>adb push hello_arm64 /data/local/tmp/hello_arm64
C:\>adb shell chmod +x /data/local/tmp/hello_arm64
C:\>adb shell /data/local/tmp/hello_arm64
Привет, архитектор компьютеров!
```

На этом наше знакомство с 32-разрядной и 64-разрядной версиями архитектуры ARM завершается.

Резюме

После прочтения этой главы вы должны получить хорошее представление о высокоуровневой организации и функциях регистров, наборов инструкций и языков ассемблера архитектур *x*86, *x*64 и ARM (32-разрядной и 64-разрядной).

Архитектуры *x*86 и *x*64 в целом отражают CISC-подход к проектированию процессоров, используя инструкции разной длины, выполнение которых может занять много тактов, длинный конвейер и (в *x*86) ограниченное количество процессорных регистров.

Архитектуры ARM, с другой стороны, реализуют RISC-подход с преимущественно однотактным выполнением инструкций, большим набором регистров и инструкциями фиксированной длины (в известной степени). Ранние версии архитектуры ARM имели конвейеры длиной всего в три этапа, хотя более в поздних поколениях число этапов значительно увеличено.

Можно ли сказать, что одна из этих архитектур лучше другой в общем смысле? Каждая из них в чем-то может быть лучше другой, и разработчики систем должны выбирать архитектуру процессора, исходя из конкретных потребностей разрабатываемой системы. Конечно, применению процессоров *x*86/*x*64 в персональных компьютерах, компьютерах бизнес-класса и серверах свойственна значительная инерция. Аналогичным образом, доминирование процессоров ARM в мобильных устройствах и встраиваемых системах имеет длинную историю. При выборе процессора для проектирования нового компьютера или мобильного устройства необходимо учитывать множество факторов, помимо чистой производительности.

В следующей главе мы рассмотрим архитектуру RISC-V. Она была разработана с чистого листа, с учетом уроков, извлеченных из истории разработки процессоров, и без какого-либо багажа, требующего поддержки устаревших конструкций десятилетней давности.

Упражнения

1. Установите бесплатный выпуск Visual Studio Community, доступный по адресу **https://visualstudio.microsoft.com/vs/community/**, на ПК с ОС Windows. После завершения установки откройте среду разработки Visual Studio IDE и выберите из меню **Tools** (Инструменты) пункт **Get Tools and Features...** (Получить инструменты и возможности...). Установите рабочую нагрузку **Desktop development with C++** (Разработка классических приложений на C++).

   В окне поиска Windows на панели задач начните вводить `Developer Command Prompt for VS 2022`. Когда приложение появится в меню поиска, выберите его, чтобы открыть командную строку.

   Создайте файл с именем `hello_x86.asm` с содержимым, показанным в листинге исходного кода в *разд. "Язык ассемблера x86" этой главы*.

   Выполните сборку программы, используя команду, приведенную в *разд. "Язык ассемблера x86" этой главы*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

2. Напишите на ассемблере *x*86 программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – 66) × × (445 + 136)] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

3. В строке поиска Windows на панели задач начните вводить `x64 Native Tools Command Prompt for VS 2022`. Когда приложение появится в меню поиска, выберите его, чтобы открыть командную строку.

   Создайте файл с именем `hello_x64.asm` с содержимым, показанным в листинге исходного кода в *разд. "Язык ассемблера x64" этой главы*.

   Выполните сборку программы, используя команду, приведенную в *разд. "Язык ассемблера" x64 этой главы*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

4. Напишите на ассемблере *x*64 программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – 66) × × (445 + 136)] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

5. Установите бесплатный пакет Android Studio IDE, доступный по адресу **https://developer.android.com/studio/**. После завершения установки откройте среду разработки Android Studio IDE, создайте новый проект и выберите из меню **Tools** (Инструменты) пункт **SDK Manager** (Диспетчер SDK). Выберите вкладку **SDK Tools** (Инструменты SDK) и установите флажок **NDK**, который может иметь обозначение **NDK (Side by side)**. Завершите установку NDK.

   Найдите следующие файлы в каталоге установки SDK (местоположение по умолчанию: `%LOCALAPPDATA%\Android`) и добавьте эти каталоги в переменную среды `PATH`: `arm-linux-androideabi-as.exe` и `adb.exe`. Подсказка: следующая команда предназначена для конкретной версии Android Studio (ваш путь может отличаться):

```
set PATH=%PATH%;%LOCALAPPDATA%\Android\Sdk\ndk\23.0.7599858\toolchains\llvm\
prebuilt\windows-x86_64\bin
```

   Создайте файл с именем `hello_arm.s` с содержимым, показанным в листинге исходного кода в *разд. "32-разрядный язык ассемблера ARM" этой главы*.

   Выполните сборку программы, используя команды, показанные в *разд. "32-разрядный язык ассемблера ARM" этой главы*.

   Включите **режим разработчика** на телефоне или планшете Android. Указания о том, как это сделать, можно найти в Интернете.

   Подсоедините Android-устройство к компьютеру с помощью USB-кабеля.

   Скопируйте исполняемый файл программы на телефон, используя команды, приведенные в *разд. "32-разрядный язык ассемблера ARM" этой главы*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

   Отключите **режим разработчика** на телефоне или планшете Android.

6. Напишите на 32-разрядном ассемблере ARM программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – 66) × (445 + 136)] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.
7. Найдите следующие файлы в каталоге установки SDK (местоположение по умолчанию: `%LOCALAPPDATA%\Android`) и добавьте эти каталоги в переменную среды `PATH`: `aarch64-linux-android-as.exe` и `adb.exe`. Подсказка: следующая команда предназначена для конкретной версии Android Studio (ваш путь может отличаться):

   ```
   set PATH=%PATH%;%LOCALAPPDATA%\Android\Sdk\ndk\23.0.7599858\toolchains\llvm\
   prebuilt\windows-x86_64\bin;%LOCALAPPDATA%\Android\ Sdk\platform-tools
   ```

   Создайте файл с именем `hello_arm64.s` с содержимым, показанным в листинге исходного кода в *разд. "64-разрядный язык ассемблера ARM" этой главы.*

   Выполните сборку программы, используя команды, показанные в *разд. "64-разрядный язык ассемблера ARM" этой главы.*

   Включите **режим разработчика** на телефоне или планшете Android. Подсоедините Android-устройство к компьютеру с помощью USB-кабеля.

   Скопируйте исполняемый файл программы на телефон, используя команды, приведенные в *разд. "64-разрядный язык ассемблера ARM" этой главы*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

   Отключите **режим разработчика** на телефоне или планшете Android.
8. Напишите на 64-разрядном ассемблере ARM программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – 66) × (445 + 136)] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

11

Архитектура и набор инструкций RISC-V

В этой главе представлена относительно новая перспективная процессорная архитектура RISC-V (произносится как *"риск файв"*) с соответствующим набором инструкций. RISC-V — это спецификация с полностью открытым исходным кодом для процессора с сокращенным набором инструкций. Были выпущены полные спецификации для набора инструкций пользовательского режима (непривилегированных) и набора привилегированных инструкций, и в настоящее время доступно широкое разнообразие аппаратных реализаций этой архитектуры. Также имеются спецификации ряда расширений набора инструкций для поддержки вычислений общего назначения, высокопроизводительных вычислений и встраиваемых решений. Представленные на рынке процессоры реализуют многие из этих расширений.

В этой главе будут рассмотрены следующие темы:

- архитектура и приложения RISC-V;
- базовый набор инструкций RISC-V;
- расширения RISC-V;
- варианты RISC-V;
- 64-разрядная архитектура RISC-V;
- стандартные конфигурации RISC-V;
- язык ассемблера RISC-V;
- реализация концепции RISC-V в **ПЛИС** (программируемая логическая интегральная схема — field-programmable gate array, FPGA).

После прочтения этой главы вы будете обладать знаниями об архитектуре и возможностях процессора RISC-V и предлагаемых отдельно расширений.

Вы ознакомитесь с основами набора инструкций RISC-V и поймете, как RISC-V можно адаптировать для поддержки различных компьютерных архитектур — от недорогих встраиваемых систем до ферм облачных серверов размером с большой склад. Вы также узнаете, как реализовать процессор RISC-V на недорогой плате ПЛИС.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Архитектура и приложения RISC-V

Архитектуру RISC-V, о выпуске которой было объявлено в 2014 г., разработали исследователи Калифорнийского университета в Беркли Юнсуп Ли (Yunsup Lee), Крсте Асанович (Krste Asanović), Дэвид Э. Паттерсон (David A. Patterson) и Эндрю Уотерман (Andrew Waterman). Эта работа последовала за четырьмя предыдущими крупными проектами архитектурных решений RISC в Калифорнийском университете в Беркли, поэтому она получила название RISC-V, где V обозначает римскую цифру "пять".

Проект RISC-V начинался с чистого листа с несколькими основными целями.

- Разработать **архитектуру набора инструкций** (instruction set architecture, ISA) RISC, подходящую для широкого спектра областей применения — от мобильных устройств с минимальным энергопотреблением до высокопроизводительных многопроцессорных облачных серверов.
- Предоставить ISA, которую может бесплатно использовать любой пользователь в любой области применения. Это контрастирует с ISA почти всех других доступных на рынке процессоров, являющихся тщательно охраняемой интеллектуальной собственностью компании, которой они принадлежат.
- Учесть уроки, извлеченные на протяжении предыдущих десятилетий проектирования процессоров, избегая неправильных поворотов и неоптимальных функций, которые в других архитектурах приходится переносить в новые поколения, чтобы поддерживать совместимость с предыдущими, иногда древними с технологической точки зрения, поколениями.
- Создать небольшую, но полнофункциональную базовую архитектуру, пригодную для применения в мобильных устройствах. Базовая ISA предоставляет минимальный набор возможностей, которые должны быть реализованы в любом процессоре RISC-V. Базовая конфигурация RISC-V — это 32-разрядный процессор с 31 регистром общего назначения.

  Все инструкции имеют длину 32 бита. Базовая ISA поддерживает сложение и вычитание целых чисел, но не включает в себя умножение и деление целых

чисел. Это позволяет избежать включения в минимальные реализации процессоров относительно дорогостоящих аппаратных средств умножения и деления для областей применения, где такие операции не требуются.

- Предоставить необязательные расширения ISA для поддержки математики с плавающей запятой, неделимых операций в памяти, а также умножения и деления.
- Предоставить дополнительные расширения ISA для поддержки привилегированных режимов выполнения, аналогичных привилегированным реализациям *x*86, *x*64 и ARM, обсуждавшимся в *главе 10*.
- Обеспечить поддержку сжатого набора инструкций, реализующего 16-разрядные версии многих 32-разрядных инструкций. В процессорах, реализующих это расширение, 16-разрядные инструкции могут свободно чередоваться с 32-разрядными инструкциями.
- Предоставить необязательные расширения ISA для поддержки 64-разрядных и даже 128-разрядных слов процессора и виртуальной памяти со страничной организацией на одноядерных и многоядерных процессорах, а также в многопроцессорных конфигурациях.

Сегодня процессоры RISC-V доступны на рынке по конкурентоспособным ценам, поэтому, учитывая уровень развития архитектуры ISA и преимущества ее бесплатного использования, можно ожидать, что в ближайшие годы доля процессоров RISC-V на рынке будет быстро расти. Доступны дистрибутивы RISC-V для Linux, которые включают в себя все средства разработки программного обеспечения, необходимые для создания и запуска приложений на компьютерах и мобильных устройствах на базе RISC-V.

Некоторые области применения, где отмечается значительный рост популярности процессоров RISC-V:

- искусственный интеллект и машинное обучение;
- мобильные устройства;
- вычисления экстремального масштаба;
- вычисления со сверхнизким энергопотреблением;
- периферийные вычисления в Интернете вещей.

Мы рассмотрим некоторые реализации архитектуры RISC-V для этих областей применения позже в этой главе.

На рис. 11.1 представлен базовый набор регистров ISA RISC-V.

Регистры имеют ширину 32 бита. Регистры общего назначения от `x1` до `x31` можно использовать без каких-либо ограничений и без каких-либо специальных функций, назначенных аппаратными средствами процессора. Регистр `x0` аппаратно запрограммирован на возврат нуля при считывании и отбрасывает любое записанное в него значение. Скоро мы увидим несколько интересных вариантов применения регистра `x0`.

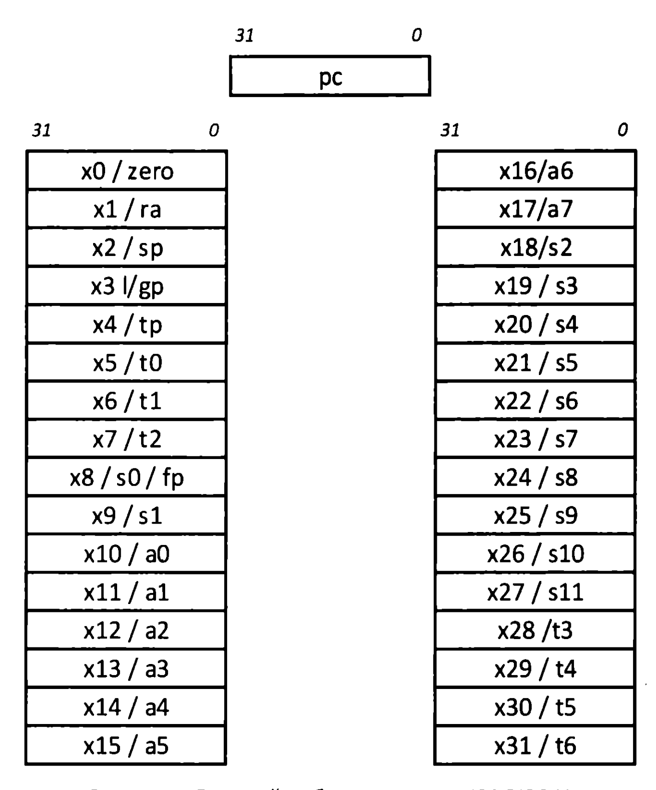


Рис. 11.1. *Базовый набор регистров ISA RISC-V*

Каждый регистр имеет одно или два альтернативных имени, показанных на рис. 11.1. Эти имена соответствуют использованию регистров в стандартном **двоичном интерфейсе прикладных программ** (application binary interface, ABI) RISC-V. ABI необходим, т. к. регистры `x1`–`x31` являются функционально взаимозаменяемыми, а для совместимости программного обеспечения требуется, чтобы мы указали, какой регистр должен служить указателем стека, какие регистры должны содержать аргументы функций, возвращаемые значения, и т. д. Обозначения регистров:

- `ra` — адрес возврата функции;
- `sp` — указатель стека;
- `gp` — глобальный указатель данных;
- `tp` — локальный указатель данных (на уровне потока);
- `t0`–`t6` — временное хранение;
- `fp` — указатель кадра для данных локального стека (на уровне функции) (этот вариант использования не является обязательным);
- `s0`–`s11` — сохраняемые регистры (если указатель кадра не используется, `x8` становится `s0`);
- `a0`–`a7` — аргументы, передаваемые в функции. Любые дополнительные аргументы передаются в стек. Возвращаемые функцией значения передаются в `a0` и `a1`.

Регистр pc содержит 32-разрядный программный счетчик, содержащий адрес текущей инструкции.

Вас может удивить тот факт, что в ISA RISC-V отсутствуют флаги процессора. Некоторые операции, которые изменяют флаги в других процессорных архитектурах, в RISC-V сохраняют результаты в регистре. Например, инструкции RISC-V для знакового (slt) и беззнакового (sltu) сравнения выполняют вычитание одного операнда из другого и устанавливают в регистре-приемнике значение 0 или 1 в зависимости от знака результата. Последующая инструкция условного ветвления может использовать значение в этом регистре, чтобы определить, какой путь выполнения кода выбрать.

Некоторые флаги, используемые в других процессорах, в RISC-V необходимо вычислять. Например, здесь нет флага переноса. Для того чтобы определить, привело ли сложение к переносу, нужно сравнить сумму и один из операндов инструкции сложения. Если эта сумма больше или равна слагаемому (для сравнения можно использовать любое слагаемое), переноса не было, в противном случае сложение породило перенос.

В большинстве базовых вычислительных инструкций ISA используется формат с тремя операндами, где первый операнд является регистром-приемником, второй операнд — регистром-источником, а третий операнд представляет собой либо регистр-источник, либо непосредственное значение. Вот пример инструкции с тремя операндами:

```
add x1, x2, x3
```

Эта инструкция добавляет содержимое регистра x2 к содержимому регистра x3 и сохраняет результат в регистре x1.

Для того чтобы избежать введения инструкций, которые не являются строго необходимыми, многим инструкциям назначены дополнительные функции, которые в других процессорных архитектурах выполняются отдельными инструкциями. Например, в RISC-V отсутствует инструкция, которая просто перемещает содержимое одного регистра в другой. Вместо этого инструкция сложения в RISC-V складывает значение из регистра-источника и непосредственное значение, равное нулю, и сохраняет результат в регистре-приемнике, получая тот же результат. Таким образом, инструкция для перемещения содержимого регистра x2 в регистр x1 выглядит так: add x1,x2,0 — она помещает значение (x2 + 0) в регистр x1.

Язык ассемблера RISC-V предоставляет несколько псевдоинструкций, использующих знакомую терминологию для реализации таких функций. Например, ассемблер переводит псевдоинструкцию mv x1,x2 в буквальную инструкцию add x1,x2,0.

Базовый набор инструкций RISC-V

Базовый набор инструкций RISC-V состоит всего из 47 инструкций. Восемь из них — это системные инструкции, которые выполняют системные вызовы и полу-

чают доступ к счетчикам производительности. Остальные 39 инструкций относятся к категориям вычислительных инструкций, инструкций потока управления и инструкций доступа к памяти. Мы рассмотрим каждую из этих категорий по очереди.

Вычислительные инструкции

Все вычислительные инструкции, кроме lui и auipc, используют формат с тремя операндами. Первый операнд — это регистр-приемник, второй — регистр-источник, а третий — либо второй регистр-источник, либо непосредственное значение. Обозначения инструкций, использующих непосредственное значение (за исключением auipc), заканчиваются буквой i. Вот эти инструкции и их функции.

- add, addi, sub — сложение и вычитание. Непосредственное значение в инструкции addi — это 12-битное значение со знаком. Инструкция sub вычитает второй операнд-источник из первого. Инструкция subi отсутствует, т. к. addi можно использовать для добавления отрицательного непосредственного значения.
- sll, slli, srl, srli, sra, srai — логический сдвиг влево и вправо (sll и srl) и арифметический сдвиг вправо (sra). При выполнении логического сдвига в освободившиеся битовые позиции вставляются нули. При арифметическом сдвиге вправо в освободившихся позициях повторяется бит знака. Количество битовых позиций для сдвига берется из младших 5 бит второго регистра-источника или из 5-битного непосредственного значения.
- and, andi, or, ori, xor, xori — выполнение указанной побитовой операции над двумя операндами-источниками. Непосредственные значения, используемые в качестве операндов, имеют ширину 12 бит.
- slt, slti, sltu, sltui — инструкции *Set if Less Than (установить, если меньше, чем)* устанавливают в регистре-приемнике значение 1, если первый операнд-источник меньше, чем второй операнд-источник. Это сравнение выполняется в дополнительном коде (slt) или для беззнаковых операндов (sltu). В качестве операндов используются 12-битные непосредственные значения.
- lui — загрузка непосредственного значения в старшие биты. Эта инструкция загружает в биты 12–31 регистра-приемника 20-битное непосредственное значение. Для записи в регистр произвольного 32-разрядного непосредственного значения требуются две инструкции: сначала lui устанавливает в битах 12–31 старшие 20 битов значения, затем addi добавляет младшие 12 битов для получения полного 32-разрядного итогового значения. lui имеет два операнда: регистр-приемник и непосредственное значение.
- auipc — добавление непосредственного значения из старших битов к значению программного счетчика и сохранение результата в регистре-приемнике. Эта инструкция добавляет 20-битное непосредственное значение к старшим 20 битам программного счетчика (ПС). В RISC-V эта инструкция реализует адресацию относительно значения ПС.

  Для получения полного 32-разрядного адреса относительно ПС auipc формирует частичный результат, после чего инструкция addi добавляет младшие 12 бит.

Инструкции потока управления

Инструкции условного ветвления выполняют сравнение двух регистров и на основании результата могут передавать управление в пределах диапазона, задаваемого знаковым 12-битным смещением адреса относительно текущего значения ПС. Предусмотрены две инструкции безусловного перехода, одна из которых (`jalr`) обеспечивает доступ ко всему 32-разрядному диапазону адресов.

- `beq`, `bne`, `blt`, `bltu`, `bge`, `bgeu` — переход при следующих условиях: равно (`beq`), не равно (`bne`), меньше чем (`blt`), меньше чем без знака (`bltu`), больше или равно (`bge`), больше или равно без знака (`bgeu`). Эти инструкции выполняют указанное сравнение между двумя регистрами и, если условие выполнено, передают управление по адресу со смещением, заданным непосредственным 12-битным значением со знаком.
- `jal` — переход и связывание. Передача управления по адресу со смещением относительно ПС, заданного 20-битным непосредственным значением со знаком, и сохранение адреса следующей инструкции (адрес возврата) в регистре-приемнике.
- `jalr` — переход, связывание и регистр. Вычисляет целевой адрес как сумму регистра-источника и знакового 12-битного непосредственного значения, затем выполняет переход по этому адресу и сохраняет адрес следующей инструкции в регистре-приемнике. Когда ей предшествует `auipc`, инструкция `jalr` может выполнить переход относительно ПС в любое место 32-битного адресного пространства.

Инструкции доступа к памяти

Инструкции доступа к памяти перемещают данные между регистром и ячейкой памяти. Первый операнд — это регистр, в который требуется загрузить значение или содержимое которого следует сохранить. Второй операнд — регистр, содержащий адрес в памяти. К адресу в регистре добавляется 12-битное непосредственное значение со знаком для получения конечного адреса, используемого для загрузки или сохранения.

Инструкции загрузки выполняют расширение знака для знаковых значений или дополнение нулями для беззнаковых значений. При загрузке значения данных меньшего размера (байта или полуслова) операция расширения знака или дополнения нулями обеспечивает заполнение остальных 32 битов в регистре-приемнике.

На работу с беззнаковыми значениями в обозначении инструкции указывает завершающий символ `u`.

- `lb`, `lbu`, `lh`, `lhu`, `lw` — загрузка в регистр-приемник 8-битного байта (`lb`), 16-битного полуслова (`lh`) или 32-битного слова (`lw`). При загрузке байтов и полуслов инструкции либо распространяют знак (`lb` и `lh`), либо дополняют нулями (`lbu` и `lhu`) остальные биты 32-битного регистра-приемника.

  Например, инструкция `lw x1, 16(x2)` загружает слово по адресу (`x2` + 16) в регистр `x1`.

- sb, sh, sw — сохранение байта (sb), полуслова (sh) или слова (sw) в область памяти, соответствующую размеру значения данных.
- fence — упорядочение доступа к памяти в многопоточном контексте. Цель этой инструкции — обеспечить согласованное представление кешированных данных в разных потоках. Она принимает два операнда: первый определяет типы предыдущих операций обращения к памяти, которые должны завершиться до выполнения инструкции fence, второй — типы последующих операций обращения к памяти, выполнение которых ограничивает инструкция fence. Типы операций, упорядочиваемых этой инструкцией, — чтение из памяти и запись в память (r и w), а также ввод и вывод через устройства ввода-вывода (i и o). Например, инструкция fence rw, rw гарантирует, что все операции чтения и записи с использованием адресов памяти, начатые до инструкции fence, завершатся до начала любых последующих операций чтения из памяти или записи в память. Эта инструкция гарантирует, что любые значения, находящиеся в блоках кеш-памяти процессора, надлежащим образом синхронизированы с памятью или устройством ввода-вывода.
- fence.i — инструкция гарантирует, что все операции сохранения в памяти инструкций будут завершены до выполнения инструкции fence.i. Данная инструкция в основном полезна в контексте самомодифицирующегося кода.

Системные инструкции

Из восьми системных инструкций одна выполняет системный вызов, одна устанавливает контрольную точку для отладчика, а остальные шесть считывают и записывают данные в **регистры управления и состояния** (control and status registers, CSR) системы. Инструкции манипулирования регистрами CSR считывают текущее значение выбранного CSR в регистр, затем обновляют CSR, записывая новое значение, обнуляя или устанавливая выбранные биты. Исходное значение для модификации CSR предоставляется в регистре или в виде 5-битного непосредственного значения. Для идентификации CSR используется 12-битный адрес. Каждая инструкция CSR выполняет чтение и запись CSR как неделимую операцию.

- ecall — выполнение системного вызова. Регистры, используемые для передачи параметров в вызов и возврата из него, определяются с помощью интерфейса ABI, а не аппаратными средствами процессора.
- ebreak — установка контрольной точки отладчика.
- csrrw, csrrwi, csrrc, csrrci, csrrs, csrrsi — считывание содержимого указанного регистра CSR в регистр-приемник и запись в него значения операнда-источника (csrrw), обнуление в регистре-приемнике битов, имеющих значение 1 в операнде-источнике (csrrc), либо установка в регистре-приемнике битов, имеющих значение 1 бита в операнде-источнике (csrrs).

  Эти инструкции принимают три операнда: первый — регистр-приемник, куда помещается значение, считанное из CSR, второй — адрес CSR, третий — регистр-источник или 5-битное непосредственное значение (суффикс i).

В базовой архитектуре RISC-V определены шесть регистров CSR, которые доступны только для чтения. Для того чтобы выполнить любую из инструкций доступа к CSR в режиме только для чтения, в качестве третьего операнда необходимо указать регистр `x0`.

Следующие регистры определяют три 64-битных счетчика производительности.

- `cycle`, `cycleh` — младшие (`cycle`) и старшие (`cycleh`) 32 бита 64-битного счетчика системных тактов, прошедших с начала отсчета времени — как правило, с момента запуска системы. Частота системного тактового сигнала может меняться, если включена функция **динамического изменения напряжения и частоты** (dynamic voltage frequency scaling, DVFS) .
- `time`, `timeh` — младшие (`time`) и старшие (`timeh`) 32 бита 64-битного счетчика тактов часов реального времени с фиксированной частотой, прошедших с начала отсчета времени — как правило, с момента запуска системы.
- `instret`, `instreth` — младшие (`instret`) и старшие (`instreth`) 32 бита 64-битного счетчика выполненных инструкций процессора. Выполненные инструкции — это те, выполнение которых завершено.

Обе 32-битные половины каждого счетчика производительности нельзя прочитать с помощью одной неделимой операции. Для предотвращения ошибок и надежного считывания показаний каждого из 64-битных счетчиков необходимо использовать следующую процедуру:

1. Считывание старших 32 бит счетчика в регистр.
2. Считывание младших 32 бит счетчика в другой регистр.
3. Считывание старших 32 бит счетчика в третий регистр.
4. Сравнение результатов первого и второго считывания старших 32 бит счетчика. Если они отличаются, возврат к *шагу 1*.

Эта процедура приведет к считыванию действительного значения счетчика, даже если счетчик продолжает работать между моментами считывания. Обычно для выполнения этой последовательности требуется, самое большее, один обратный переход к *шагу 4*.

Псевдоинструкции

Архитектура RISC-V имеет сокращенный набор инструкций, в котором отсутствуют инструкции нескольких типов, включенных в наборы инструкций, которые мы исследовали в предыдущих главах. Функции многих из этих уже знакомых инструкций могут быть выполнены с помощью инструкций RISC-V, хотя, возможно, не всегда интуитивно понятным образом.

Ассемблер RISC-V поддерживает несколько псевдоинструкций, каждая из которых преобразуется в одну или несколько инструкций RISC-V, обеспечивающих функциональность, предположительно ожидаемую в наборе инструкций процессора об-

щего назначения. В табл. 11.1 представлено несколько наиболее полезных псевдоинструкций RISC-V.

Таблица 11.1. *Псевдоинструкции RISC-V*

Псевдоинструкция	Инструкции RISC-V	Функция
nop	addi x0, x0, 0	Без операции
mv rd, rs	addi rd, rs, 0	Копирование rs в rd
not rd, rs	xori rd, rs, -1	rd = NOT rs
neg rd, rs	sub rd, x0, rs	rd = -rs
j offset	jal x0, offset	Безусловный переход
jal offset	jal x1, offset	Ближний вызов функции (20-битное смещение)
call offset	auipc x1, offset [31:12] + offset [11] jalr x1, offset [11:0] (x1)	Дальний вызов функции (32-битное смещение)
ret	jalr x0, 0 (x1)	Возврат из функции
beqz rs, offset	beq rs, x0, offset	Переход, если равно нулю
bgez rs, offset	bge rx, x0, offset	Переход, если больше или равно нулю
bltz rs, rt, offset	blt rs, x0, offset	Переход, если меньше нуля
bgt rs, rt, offset	blt rt, rs, offset	Переход, если больше
ble rs, rt, offset	bge rt, rs, offset	Переход, если меньше или равно
fence	fence iorw, iorw	Выравнивание всех операций обращения к памяти и операций ввода-вывода через память
csrr rd, csr	csrrw rd, csr, x0	Считывание регистра CSR
li rd, immed	addi rd, x0, immed	Загрузка 12-битного непосредственного значения
li rd, immed	lui rd, immed [31:12] + immed [11] addi rd, x0, immed [11:0]	Загрузка 32-битного непосредственного значения
la rd, symbol	auipc rd, delta [31:12] + delta [11] addi rd, rd, delta [11:0]	Загрузка адреса symbol, где delta=(symbol-pc)
lw rd, symbol	auipc rd, delta [31:12] + delta [11] lw rd, rd, delta [11:0] (rd)	Загрузка слова по адресу symbol, где delta=(symbol-pc)
sw rd, symbol, rt	auipc rt, delta [31:12] + delta [11] sw rd, rd, delta [11:0] (rt)	Сохранение слова по адресу symbol, где delta=(symbol-pc)

В табл. 11.1: `rd` — регистр-приемник, `rs` — регистр-источник, `csr` — регистр управления и состояния, `symbol` — абсолютный адрес данных, `offset` — адрес инструкции относительно ПС.

Инструкции, объединяющие старшие 20 бит адреса или непосредственного значения с непосредственным значением, содержащим младшие 12 бит, должны выполнять шаг, обратный эффекту расширения знака бита 11 младшего 12-битного значения во второй инструкции каждой последовательности. Это необходимо, т. к. непосредственное значение в инструкции `addi` всегда рассматривается как значение со знаком. Перед добавлением 12-битного непосредственного значения к старшим 20 битам выполняется распространение старшего бита этого значения до бита 31.

Следующий пример демонстрирует эту проблему и ее решение. Предположим, что требуется загрузить значение `0xFFFFFFFF` в регистр с помощью инструкций `lui` и `addi`, и просто сложим старшую и младшую части, как показано здесь:

```
lui x1, 0xFFFFF # x1 теперь равно 0xFFFFF000
addi x1, x1, 0xFFF
```



Следует отметить один момент, касающийся кода ассемблера в этой главе: в ассемблере RISC-V начало комментария обозначается символом #.

Инструкция `addi` выполняет распространение знакового разряда в `0xFFF` до `0xFFFFFFFF` перед добавлением к `0xFFFFF000`. Результатом сложения будет `0xFFFFEFFF`, а это не то, что мы хотим получить. Добавление бита 11 из младших 12 бит к старшим 20 битам исправит это, как показано в следующем блоке кода:

```
lui x1, 0xFFFFF+1 # Добавим бит 11; x1 теперь равен 0x00000000
addi x1, x1, 0xFFF
```

Теперь результат равен `0xFFFFFFFF`, что является правильным значением. Эта процедура будет работать и для любого другого числового значения. Если бит 11 окажется равным 0, то к старшим 20 битам ничего добавлено не будет.

Уровни привилегий

В архитектуре RISC-V определены три уровня привилегий, на которых может работать поток:

- пользователь (`U`);
- супервизор (`S`);
- машина (`M`).

Все реализации RISC-V должны поддерживать режим M — наиболее привилегированный уровень, который предоставляет доступ ко всем функциям системы. Этот режим активен после перезагрузки системы. Код в простой встраиваемой системе может полностью выполняться в режиме M.

В несколько более сложном случае процесс безопасной загрузки может выполняться на уровне привилегий M, загружая, проверяя и запуская приложение, которое выполняется в режиме U. Такой подход позволяет построить безопасное встраиваемое решение.

В дополнение к обязательному уровню M процессор RISC-V может поддерживать уровень U или оба уровня — S и U. Система, работающая под управлением операционной системы общего назначения, использует режим S и режим U так же, как режимы ядра и пользователя процессоров и операционных систем, рассмотренных в предыдущих главах.

В RISC-V приложения, работающие в режиме U, запрашивают системные сервисы с помощью инструкции `ecall` (вызов среды исполнения), генерирующей исключение, обрабатываемое на уровне S. Структура привилегий RISC-V напрямую поддерживает современные операционные системы, такие как Linux.

На каждом из трех уровней привилегий определены отдельные наборы регистров CSR для настройки конфигурации, управления и мониторинга системы. В зависимости от уровня привилегий запущенного потока и уровня CSR, поток может иметь доступ к CSR в режиме чтения-записи или только чтения либо не иметь доступа. Потоки на более высоких уровнях привилегий могут получить доступ к CSR на более низких уровнях привилегий.

На уровне привилегий S в RISC-V поддерживается виртуальная память со страничной организацией с 32-битным адресным пространством, разделенным на страницы по 4 Кбайт. 32-битный виртуальный адрес разделен на 20-битный номер виртуальной страницы и 12-битное смещение страницы.

Для 64-разрядной среды RISC-V определены две дополнительные конфигурации виртуальной памяти. Первая использует 39-битное адресное пространство, поддерживающее 512 Гбайт виртуальной памяти.

Для приложений, требующих еще большего числа виртуальных адресов, доступно 48-битное адресное пространство, предоставляющее доступ к 256 Тбайт виртуальной памяти. 48-битная конфигурация предлагает гораздо больше памяти, чем 39-битная, однако она также требует дополнительного объема памяти для хранения таблиц страниц, а обработка этих таблиц занимает больше времени.

Следующие инструкции поддерживают привилегированные уровни выполнения.

- `mret`, `sret`, `uret` — возврат из обработчика исключений, инициированного инструкцией `ecall`. Каждая из этих инструкций может быть выполнена на уровне привилегий, обозначенном первой буквой в их названиях, или на более высоком. Выполнение одной из этих инструкций для уровня привилегий ниже, чем у текущего потока, приведет к возврату из исключения, инициированного на более низком уровне.

- `wfi` — ожидание прерывания. Эта инструкция запрашивает приостановку текущего потока до тех пор, пока не появится прерывание, требующее обслуживания. Спецификация RISC-V требует, чтобы эта инструкция служила только подсказкой, поэтому в конкретной реализации инструкция `wfi` может обрабатываться как пустая и не приводить к приостановке потока. Поскольку процессор может обрабатывать `wfi` как пустую операцию, код, следующий за инструкцией `wfi`, должен явно проверять наличие ожидающих прерываний, нуждающихся в обработке. Эта последовательность обычно возникает внутри цикла.
- `sfence.vma` — сброс данных таблиц страниц виртуальной памяти из кеша в память. Буква `s` в начале обозначения инструкции указывает на то, что она предназначена для использования на уровне привилегий супервизора.

RISC-V определяет дополнительные инструкции и регистры CSR, поддерживающие виртуализацию и гипервизор, управляющий виртуальной средой. Виртуализация RISC-V будет рассмотрена в *главе 12*.

Расширения RISC-V

Описанный выше набор инструкций называется **RV32I** (RISC-V 32-bit Integer instruction set), что расшифровывается как **набор инструкций RISC-V для 32-разрядных целых чисел**. Архитектура набора инструкций RV32I предоставляет полный и практичный набор инструкций для многих целей, однако в нем отсутствует ряд функций и возможностей, доступных в других популярных процессорах, таких как *x*86 и ARM.

Для поэтапного добавления возможностей к базовому набору инструкций с учетом требований совместимости в архитектуре RISC-V предусмотрены расширения. Разработчики процессоров RISC-V могут выборочно включать расширения в свои проекты, чтобы добиться оптимального соотношения между размером микросхемы, возможностями системы и производительностью.

Такие же гибкие возможности проектирования доступны и разработчикам недорогих систем на базе ПЛИС. Подробнее о реализации процессора RISC-V на основе ПЛИС мы поговорим позже в этой главе. Основные расширения, которые мы рассмотрим, получили названия в виде букв `M`, `A`, `C`, `F` и `D`. Мы также упомянем некоторые другие доступные расширения.

Расширение M

Расширение `M` архитектуры RISC-V добавляет к базовому набору инструкций RV32I функции целочисленного умножения и деления. В состав этого расширения включены следующие инструкции.

- `mul` — перемножение двух 32-битных регистров и сохранение младших 32 бит результата в регистре-приемнике.

- mulh, mulhu, mulhsu — перемножение двух 32-битных регистров и сохранение старших 32 бит результата в регистре-приемнике. В этих инструкциях множители рассматриваются как оба со знаком (mulh), оба без знака (mulhu) или как rs1 со знаком и rs2 без знака (mulhsu). rs1 — это первый регистр-источник инструкции, а rs2 — второй.
- div, divu — деление двух 32-битных регистров и округление результата в направлении нуля с использованием знаковых (div) или беззнаковых (divu) операндов.
- rem, remu — возвращение остатка от деления, выполненного с помощью инструкции div или divu.

Деление на ноль не вызывает выдачу исключения. Для того чтобы обнаружить деление на ноль, код должен проверить делитель и перейти к соответствующему обработчику, если делитель равен нулю.

Расширение A

Расширение A набора инструкций RISC-V содержит неделимые операции чтения-изменения-записи для поддержки многопоточной обработки в общей памяти.

Инструкции загрузки с резервированием (lr.w) и сохранения с условием (sc.w) работают в паре, выполняя чтение ячейки памяти, после чего следует запись в ту же ячейку в неделимой последовательности. Инструкция загрузки с резервированием резервирует адрес памяти во время загрузки. Если во время действия этого резервирования другой поток записывает данные в ту же ячейку, резервирование отменяется.

Инструкция сохранения с условием возвращает значение, показывающее, насколько успешным было выполнение неделимой операции. Если резервирование сохраняет действие (другими словами, если по целевому адресу не было промежуточной записи), инструкция сохранения с условием записывает содержимое регистра в память и возвращает ноль, указывая на успешное выполнение.

Если резервирование было отменено, инструкция сохранения с условием не изменяет содержимое ячейки памяти и возвращает ненулевое значение, указывающее на неудачу операции сохранения. Операции загрузки с резервированием и сохранения с условием реализуют следующие инструкции.

- lr.w — загрузка в регистр данных из ячейки памяти и резервирование этой ячейки.
- sc.w — условное сохранение содержимого регистра в ячейку памяти. Инструкция записывает ноль в регистр-приемник, если операция прошла успешно и запись в ячейку памяти была выполнена, или помещает в этот регистр ненулевое значение, если резервирование ячейки было отменено. Если резервирование было отменено, эта инструкция не изменяет содержимое ячейки памяти.

Инструкции **неделимых операций в памяти** (atomic memory operation, AMO) в неделимой последовательности загружают слово из ячейки памяти в регистр-

приемник, выполняют двоичную операцию между загруженным значением и `rs2`, после чего сохраняют результат в памяти по адресу данной ячейки. Операции AMO реализуют следующие инструкции.

- `amoswap.w` — неделимый обмен значениями между `rs2` и ячейкой памяти `rs1`.
- `amoadd.w` — неделимое добавление `rs2` к ячейке памяти `rs1`.
- `amoand.w`, `amoor.w`, `amoxor.w` — выполнение неделимых операций И, ИЛИ и "исключающее ИЛИ" над содержимым регистра `rs2` с записью результата в ячейку памяти `rs1`.
- `amomin.w`, `amominu.w`, `amomax.w`, `amomaxu.w` — выполнение неделимых операций выбора минимального или максимального знакового или беззнакового (инструкции с суффиксом `u`) значений над `rs2` с записью результата в ячейку памяти `rs1`.

Расширение *C*

Расширение `C` набора инструкций RISC-V реализует сжатые инструкции с целью свести к минимуму объем памяти для хранения инструкций и уменьшить трафик по шине, требуемый для извлечения инструкций.

Все рассмотренные ранее инструкции RV32I имеют длину 32 бита. Расширение `C` предлагает альтернативные 16-разрядные представления многих наиболее часто используемых инструкций RV32I. Каждая сжатая инструкция эквивалентна одной полноразмерной инструкции. Переключение режимов не требуется, и это означает, что в программах можно свободно смешивать 32-разрядные инструкции RV32I и сжатые 16-разрядные инструкции.

На самом деле программистам на ассемблере даже не нужно предпринимать какие-либо действия, чтобы указать, следует ли генерировать инструкцию в сжатом виде. Ассемблер и компоновщик способны прозрачно выдавать сжатые инструкции для уменьшения размера кода, где это возможно, и в большинстве случаев — без снижения производительности выполнения программ.

При работе с процессорами и наборами инструментов разработки программного обеспечения, поддерживающими расширение `C` RISC-V, преимущества сжатых инструкций сразу же становятся доступны разработчикам, использующим язык ассемблера, а также тем, кто работает с языками более высокого уровня.

Расширения *F* и *D*

Расширения `F` и `D` набора инструкций RISC-V обеспечивают аппаратную поддержку арифметики с плавающей запятой одинарной точности (`F`) и двойной точности (`D`) в соответствии со стандартом IEEE 754. Расширение `F` добавляет в архитектуру 32 регистра с плавающей запятой от `f0` до `f31` и регистр управления и состояния с именем `fcsr`. Все эти регистры имеют разрядность 32 бита. Это расширение включает в себя набор инструкций с плавающей запятой, которые соответствуют определениям одинарной точности в стандарте IEEE 754-2008.

Большинство инструкций с плавающей запятой работают с регистрами с плавающей запятой. В этом расширении реализованы инструкции передачи данных для загрузки данных из памяти в регистры с плавающей запятой, сохранения содержимого регистров с плавающей запятой в памяти и перемещения данных между регистрами с плавающей запятой и регистрами целых чисел.

Расширение `D` увеличивает разрядность регистров `f0`–`f31` до 64 битов. В этой конфигурации каждый регистр `f` может содержать 32- или 64-битное значение. Добавлены инструкции с плавающей запятой, соответствующие определениям двойной точности в стандарте IEEE 754-2008. Расширение `D` требует наличия расширения `F`.

Другие расширения

В следующем списке приведены описания ряда дополнительных расширений архитектуры RISC-V, которые уже определены, находятся в разработке или, по крайней мере, рассматриваются для будущей разработки.

- **Архитектура RV32E.** На самом деле это не расширение, а скорее модифицированная архитектура, целью создания которой было снижение требований к аппаратным средствам процессора ниже уровня требований набора инструкций RV32I для самых маленьких встраиваемых систем. Единственным различием между RV32I и RV32E является уменьшение количества целочисленных регистров до 15. Ожидается, что это изменение уменьшит занимаемую процессором площадь кристалла и энергопотребление примерно на 25% по сравнению с аналогичным процессором RV23I. `x0` остается выделенным нулевым регистром. Уменьшение количества регистров вдвое освобождает 1 бит в каждом спецификаторе регистра в инструкции. Эти биты гарантированно останутся незадействованными в будущих версиях и, таким образом, доступны для использования в пользовательских расширениях инструкций.
- **Расширение Q** поддерживает 128-битные арифметические операции с плавающей запятой учетверенной точности согласно определениям стандарта IEEE 754-2008.
- **Расширение L** поддерживает десятичные арифметические операции с плавающей запятой согласно определениям стандарта IEEE 754-2008.
- **Расширение B** поддерживает побитовые операции, такие как вставка, извлечение и проверка состояния отдельных битов.
- **Расширение J** поддерживает динамически транслируемые языки, такие как Java и JavaScript.
- **Расширение T** поддерживает транзакции с памятью, состоящие из неделимых операций по нескольким адресам.
- **Расширение P** предоставляет упакованные инструкции **SIMD (с одним потоком инструкций и множеством потоков данных)** для операций с плавающей запятой в малых системах RISC-V.

- **Расширение V** поддерживает параллельные или векторные операции с данными. Расширение V не определяет длины векторов данных, это решение остается за разработчиками архитектуры процессора RISC-V. Типичная реализация расширения V может поддерживать 512-битные векторы данных, хотя в настоящее время доступны реализации с длиной вектора до 4096 бит.
- **Расширение N** обеспечивает поддержку обработки прерываний и исключений на уровне привилегий U.
- **Расширение Zicsr** обеспечивает выполнение неделимых операций чтения-изменения-записи в системных регистрах CSR. Эти инструкции описаны в *разд. "Системные инструкции" ранее в этой главе*.
- **Расширение Zifencei** определяет инструкцию `fence.i`, описанную в *разд. "Инструкции доступа к памяти" ранее в этой главе*.

В следующем разделе рассматриваются некоторые из доступных в настоящее время вариантов процессоров RISC-V.

Варианты RISC-V

Некоторые области применения, где получили значительное распространение различные варианты архитектуры RISC-V:

- **Искусственный интеллект и машинное обучение.** Компания Esperanto Technologies разработала **систему на кристалле (SoC)** содержащую более 1000 процессоров RISC-V и разместила шесть таких чипов на одной плате PCIe. Это решение оптимизировано для высокопроизводительных рабочих нагрузок, выдающих рекомендации на основе машинного обучения в крупных центрах обработки данных при минимальном энергопотреблении.
- **Встраиваемые системы.** Ядро Efinix VexRiscv — это программный процессор, предназначенный для реализации на ПЛИС. В VexRiscv используется набор инструкций RV32I с расширениями M и C.

  Реализация в виде полнофункциональной SoC-системы включает в себя процессор, память и выбираемый набор интерфейсов ввода-вывода, таких как входы-выходы общего назначения, таймеры, последовательные интерфейсы и интерфейсы межчиповой связи, например **последовательный периферийный интерфейс** (serial peripheral interface, SPI).
- **Вычисления экстремального масштаба.** В настоящее время предпринимается ряд усилий по разработке систем **высокопроизводительных вычислений** (high-performance computing, HPC), которые обычно называют суперкомпьютерами, основанных на процессорах RISC-V. Среди конкретных целей этих проектов — повышение энергоэффективности процессорных ядер и расширение возможностей векторной обработки на уровне аппаратных средств процессора.

- **Вычисления со сверхнизким энергопотреблением.** Компания Micro Magic предлагает 64-разрядное ядро RISC-V с заявленным энергопотреблением всего 0,01 Вт при работе на тактовой частоте 1 ГГц. Такой уровень энергопотребления позволит устройствам с аккумуляторным питанием работать без подзарядки в течение нескольких дней, недель или месяцев.
- **Периферийные вычисления в Интернете вещей.** Интеллектуальным устройствам, работающим в рамках **Интернета вещей** (Internet of Things, IoT), может потребоваться высокая вычислительная мощность для решения таких задач, как шифрование данных и формирование логических выводов на основе методов искусственного интеллекта. При этом они должны работать в условиях жестких ограничений по энергопотреблению, определяемых такими факторами, как время работы от аккумулятора и скромные возможности охлаждения. Некоторые примеры устройств Интернета вещей: камеры дверных звонков, процессоры голосовых команд, такие как Amazon Alexa, и розетки питания с поддержкой Wi-Fi. Как и в предыдущих примерах, цели создания этих процессоров заключаются в достижении максимальной производительности при одновременном ограничении энергопотребления до приемлемого уровня.

Это лишь некоторые из областей применения, где архитектура RISC-V завоевывает популярность. Можно ожидать, что расширение использования RISC-V в этих и других областях продолжится.

Следующий раздел посвящен расширению базовой архитектуры набора инструкций RISC-V до 64 битов.

64-разрядная архитектура RISC-V

В этой главе обсуждались 32-разрядная архитектура и набор инструкций RV32I, а также ряд важных расширений. Набор инструкций RV64I расширяет RV32I до 64-разрядной архитектуры. Как и в RV32I инструкции имеют разрядность 32 бита. В действительности набор инструкций RV64I почти полностью совпадает с RV32I, за исключением следующих существенных различий.

- Разрядность регистров для целочисленных вычислений увеличена до 64 бит.
- Длина адресов увеличена до 64 бит.
- Размерность битового сдвига в опкодах инструкций увеличена с 5 до 6 бит.
- Предусмотрено несколько новых инструкций для работы с 32-разрядными значениями в том же стиле, который применялся в RV32I. Потребность в этих инструкциях обусловлена тем, что большинство инструкций в RV64I работают с 64-разрядными значениями, и существует много ситуаций, в которых требуется эффективная работа с 32-разрядными значениями. Опкоды этих ориентированных на слова инструкций имеют суффикс `W`. Инструкции с суффиксом `W` выдают 32-разрядные результаты со знаком. Для заполнения

64-разрядного регистра-приемника эти 32-разрядные значения дополняются знаковым битом (даже если они являются беззнаковыми значениями). Другими словами, бит 31 каждого результата копируется в биты 32–63.

В RV64I определены следующие новые инструкции.

- addw, addiw, subw, sllw, slliw, srlw, srliw, sraw, sraiw — эти инструкции выполняются почти идентично инструкциям из RV32I с теми же именами, за исключением суффикса W. Они работают с 32-разрядными операндами и выдают 32-разрядные результаты. Результаты дополняются знаковым битом до 64 бит.
- ld, sd — загрузка и сохранение 64-разрядного **двойного слова**. Это 64-разрядные версии инструкций lw и sw в наборе инструкций RV32I.

Остальные инструкции из RV32I выполняют те же функции и в RV64I, за исключением того, что адреса и регистры имеют длину 64 бита. В обоих наборах инструкций используются одни и те же опкоды — как в исходном коде ассемблера, так и в двоичном машинном коде.

В следующем разделе мы рассмотрим некоторые представленные на рынке стандартные 32-разрядные и 64-разрядные конфигурации RISC-V. Каждая из них состоит из базовой архитектуры набора инструкций (ISA), дополненной выбранными расширениями.

Стандартные конфигурации RISC-V

Наборы инструкций RV32I и RV64I предоставляют базовый набор возможностей, который подходит в основном для проектов небольших встраиваемых систем. Для правильной и эффективной работы систем, которые должны поддерживать многопоточность, несколько уровней привилегий и операционные системы общего назначения, требуются соответствующие расширения RISC-V.

Минимальная конфигурация RISC-V, рекомендуемая для определения цели разработки приложения, состоит из базовой архитектуры набора инструкций RV32I или RV64I, дополненной расширениями I, M, A, F, D, Zicsr и Zifencei. Для обозначения этой комбинации функций используется буква G: RV32G или RV64G. Многие конфигурации G дополнительно поддерживают расширение сжатых инструкций, получая при этом обозначение RV32GC или RV64GC.

Во встраиваемых решениях распространенной конфигурацией является RV32IMAC, предоставляющая базовый набор инструкций наряду с функциями умножения/деления, неделимыми операциями и поддержкой сжатых инструкций.

Эти сокращенные описания возможностей процессора часто используются в маркетинговых материалах о процессорах RISC-V.

В следующем разделе представлена программа на языке ассемблера RISC-V.

Язык ассемблера RISC-V

Следующий пример кода на ассемблере RISC-V представляет собой законченное приложение, работающее на процессоре RISC-V:

```
.section .text
.global main

main:
    # Резервирование места в стеке и сохранение адреса возврата
    addi    sp, sp, -16
    sd      ra, 0(sp)

    # Печать сообщения с помощью функции puts библиотеки Си
1:  auipc   a0, %pcrel_hi(msg)
    addi    a0, a0, %pcrel_lo(1b)
    jal     ra, puts

# Восстановление адреса возврата и sp, возврат к месту вызова
    Ld      ra, 0(sp)
    addi    sp, sp, 16
    jalr    zero, ra, 0

.section .rodata
msg:
    .asciz "Привет, архитектор компьютеров!\n"
```

Эта программа выводит следующее сообщение в окне консоли, после чего завершает работу:

```
Привет, архитектор компьютеров!
```

Ниже описаны некоторые интересные моменты в этом коде ассемблера.

- Директивы `%pcrel_hi` и `%pcrel_lo` выбирают старшие 20 бит (`%pcrel_hi`) и младшие 12 бит (`%pcrel_lo`) относительного адреса (со смещением относительно программного счетчика) метки, предоставленной в качестве аргумента. Сочетание инструкций `auipc` и `addi` помещает адрес строки сообщения в `a0`.
- `1:` — это локальная метка. При ссылке на локальную метку к ее имени добавляется буква `b` для указания вышестоящей метки (в обратном направлении) или буква `f` для указания нижестоящей метки (в прямом направлении). Директивы `%pcrel_hi` и `%pcrel_lo` работают в паре: локальная метка `1:` преобразует младшие 12 бит смещения в адрес сообщения `msg`.

В следующем разделе мы запустим некоторый код в полнофункциональном процессоре RISC-V, реализованном в ПЛИС.

Реализация концепции RISC-V в ПЛИС

Весь исходный код, интеллектуальную собственность на конструкцию аппаратных средств процессора и средства разработки, необходимые для создания и построения полнофункционального процессора RISC-V на основе недорогой ПЛИС (программируемой логической интегральной схемы), можно найти в свободном доступе в Интернете. В этом разделе представлены общий обзор архитектуры RISC-V с открытым исходным кодом и описание действий по ее внедрению в ПЛИС. Общая стоимость оборудования для выполнения этой задачи составляет менее 200 долларов США.

В качестве ПЛИС для реализации RISC-V в этом примере выбрана плата Digilent Arty A7-35T, которую можно приобрести здесь **https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and- hobbyists/**. На момент написания книги плата Arty A7-35T стоила 129 долларов США.

Arty A7-35T содержит ПЛИС Xilinx Artix-7 XC7A35TICSG324-1L, которую можно запрограммировать для реализации процессора RISC-V. Особенности ПЛИС XC7A35TICSG324-1L:

- 5200 логических секций;
- 1600 из них можно использовать для создания 64-разрядной оперативной памяти;
- 41 600 триггеров, каждая логическая секция содержит восемь триггеров;
- 90 секций DSP поддерживают высокопроизводительное выполнение операций умножения с накоплением для DSP;
- 400 Кбит распределенной оперативной памяти;
- 1800 Кбит общей оперативной памяти.

Для реализации комбинационной логики в архитектуре ПЛИС Artix-7 используются **таблицы поиска** (LookUp Table, LUT). Каждая таблица поиска Artix-7 имеет шесть входных сигналов и один выходной сигнал, где каждый сигнал представляет собой 1 бит данных. Одна таблица поиска может представлять любую схему без обратной связи, состоящую из вентилей И, ИЛИ, НЕ и "исключающее ИЛИ", работающих с шестью входными сигналами путем простого сохранения результатов каждой входной комбинации в виде бита в небольшом ПЗУ. При наличии шести входных битов это ПЗУ содержит 64 (2^6) бита данных, адресуемых шестью входными сигналами. При желании каждую таблицу поиска также можно представить в виде двух 32-разрядных таблиц поиска, работающих на пяти общих входах с двумя выходными битами. Как вариант, выход таблицы поиска можно сохранить в триггере.

Логическая секция содержит четыре таблицы поиска и восемь триггеров плюс дополнительный мультиплексор и логику арифметического переноса. Четыре из восьми триггеров в секции можно настроить как защелки. На основе любой из

1600 секций, поддерживающих возможность создания 64-разрядной оперативной памяти, можно также реализовать 32-разрядный регистр сдвига или два 16-разрядных регистра сдвига.

Низкоуровневые таблицы поиска и другие средства, реализуемые несколькими тысячами логических секций, представляют собой исходные элементы, необходимые для сборки полнофункционального процессора RISC-V и периферийных устройств в рамках одной ПЛИС. Процесс программирования ПЛИС заключается в соединении компонентов ПЛИС для формирования сложного цифрового устройства, определенного с помощью языка описания аппаратных средств.

С точки зрения разработчика системы нет необходимости разбираться в деталях внутреннего механизма ПЛИС Xilinx. Разработчик действует на уровне языка описания аппаратных средств. Такой инструмент, как Vivado, представленный в решениях к упражнениям в *главе 2*, переводит код на языке описания аппаратных средств (обычно VHDL или Verilog, хотя оригинальный проект RISC-V реализован на языках Chisel и Scala) в скомпилированный формат, подходящий для программирования ПЛИС.

Основные вопросы, требующие внимания разработчика в отношении ПЛИС, заключаются в том, чтобы проект системы учитывал ограничения ресурсов ПЛИС, а итоговая реализация работала с приемлемой производительностью. В данном примере объем ресурсов, предлагаемых ПЛИС XC7A35TICSG324-1L, более чем достаточен для реализации процессора RISC-V.

Для разработки и запуска программ на RISC-V-процессоре Arty A7-35T также нужен недорогой аппаратный отладчик. Отладчик Olimex ARM-TINY-USB-H можно приобрести по цене 47,65 доллара США здесь: **https://www.digikey.com/product-detail/en/olimex-ltd/ARM-USB-TINY-H/1188-1013-ND/3471388**. Вам также понадобятся провода для подсоединения отладчика к плате Arty A7-35T. Их можно приобрести по цене 3,95 доллара США здесь: **https://www.adafruit.com/product/826**. Наконец, для подсоединения процессора Arty A7-35T к главному компьютеру через разъем MicroUSB-B нужен USB-кабель. Все программное обеспечение и проектные данные, необходимые для реализации RISC-V-процессора в Arty, можно бесплатно загрузить из Интернета.

Процессор, который мы реализуем на основе Arty A7-35T, — это *Freedom E310 Arty*, открытая реализация ядра RV32IMAC с поддержкой обработки прерываний. Для периферийных устройств предусмотрены 16 сигналов **входов-выходов общего назначения** (general purpose I/O, GPIO) и последовательный порт.

Процессор Freedom E310 предоставляется в виде исходного кода и поэтому может быть изменен пользователями, которые хотят реализовать собственные версии процессора. В аппаратном коде процессора RISC-V применяются языки описания аппаратных средств Chisel и Scala.

Chisel — это специализированный язык, предназначенный для разработки сложных цифровых аппаратных устройств, таких как система на кристалле (SoC). Chisel запускается поверх **Scala**, современного языка программирования общего назначе-

ния, поддерживающего парадигмы функционального и объектно-ориентированного программирования. Scala — это полностью объектно-ориентированный язык, в котором каждое значение — это объект. Он также является функциональным языком в том смысле, что каждая функция является значением. Scala компилируется в байт-код Java и запускается на стандартной виртуальной машине Java. Программы на языке Scala могут напрямую использовать любую из тысяч доступных библиотек Java.

ПОДДЕРЖКА ПОЛЬЗОВАТЕЛЬСКИХ РАСШИРЕНИЙ В RISC-V



Архитектура RISC-V явно поддерживает пользовательские версии в виде настраиваемых опкодов, сопроцессоров и других модификаций, при условии, что они совместимы с правилами адаптации RISC-V к требованиям пользователя. Взяв за основу открытую архитектуру RISC-V, вы можете реализовать пользовательские модификации, которые гарантированно останутся совместимыми с будущими версиями стандартов и расширений RISC-V.

При проектировании сложных цифровых систем некоторых категорий языки Chisel и Scala благодаря своей высокоуровневой природе сегодня являются более предпочтительными по сравнению с традиционными языками описания аппаратных средств, такими как VHDL и Verilog. Любая схема, которую можно спроектировать в Chisel, также может быть спроектирована и в VHDL, однако использование Chisel имеет некоторые важные преимущества. Например, процесс компиляции преобразует код на Chisel/Scala в форму, называемую **гибким промежуточным представлением для RTL** (Flexible Intermediate Representation for RTL, FIRRTL), где **RTL** (register transfer level) означает **уровень межрегистровой передачи**. RTL — это уровень абстракции в языках описания аппаратных средств синхронных схем, таких как VHDL. Используя бесплатные инструменты, можно оптимизировать представление схемы в FIRRTL, что позволит создать более производительную реализацию ПЛИС по сравнению с сопоставимым решением на VHDL или Verilog.

Одним из способов оценить разницу между Chisel и VHDL/Verilog является аналогичное различие между языками программирования Python и C. Вы можете реализовать функциональный эквивалент любой программы Python на C, однако программы на Python могут в нескольких строках кода выразить функциональность гораздо более высокого уровня, чем программа аналогичного размера на C.

Мы можем сравнить код Chisel с примером VHDL, приведенным в *разд. "Языки описания аппаратных средств" главы 2*. Рассмотрим версию одноразрядного полного сумматора на VHDL, представленную в этой главе и показанную в следующем коде:

```
-- Загрузка стандартных библиотек

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
-- Определение входов и выходов полного сумматора

entity FULL_ADDER is
  port (
    A     : in    std_logic;
    B     : in    std_logic;
    C_IN  : in    std_logic;
    S     : out   std_logic;
    C_OUT : out   std_logic
  );
end entity FULL_ADDER;

-- Определение поведения полного сумматора

architecture BEHAVIORAL of FULL_ADDER is

begin

  S      <= (A XOR B) XOR C_IN;
  C_OUT  <= (A AND B) OR ((A XOR B) AND C_IN);

end architecture BEHAVIORAL;
```

Эквивалент полного сумматора на языке Chisel показан в следующем блоке кода:

```
import chisel3._

class FullAdder extends Module {
  val io = IO(new Bundle {
    val a     = Input(UInt(1.W))
    val b     = Input(UInt(1.W))
    val c_in  = Input(UInt(1.W))
    val s     = Output(UInt(1.W))
    val c_out = Output(UInt(1.W))
  })
  io.s := (io.a ^ io.b) ^ io.c_in
  io.c_out := (io.a & io.b) | ((io.a ^ io.b) & io.c_in)
}
```

В этом коде Chisel пакет `IO` определяет входы и выходы модуля. Аргумент каждого параметра `Input` и `Output` определяет тип данных (`UInt`) и битовую ширину (`1.W` указывает, что каждый входной и выходной сигнал имеет ширину 1 бит).

Этот простой пример не способен продемонстрировать весь спектр преимуществ разработки сложных схем в Chisel, однако он показывает, что на уровне подробной реализации этот код не слишком отличается от VHDL. Здесь мы не будем углубляться в детали Chisel. Для получения дополнительной информации обратитесь к репозиторию Chisel по адресу **https://github.com/freechipsproject/chisel3**.

Процесс создания процессора RISC-V и его программирования на плате Arty A7-35T состоит из следующих шагов:

1. Трансляция кода Chisel и Scala в формат FIRRTL.
2. Трансляция FIRRTL в Verilog.
3. Компиляция Verilog в образ ПЛИС.
4. Загрузка образа ПЛИС в плату Arty A7-35T.

Пошаговое руководство по реализации процессора RISC-V на плате Arty A7-35T можно найти на форуме Digi-Key Electronics TechForum по адресу **https://forum.digikey.com/t/digilent-arty-a7-with-xilinx-artix-7-implementing-sifive-fe310-risc-v**.

Когда вы загрузите образ RISC-V в плату Arty, к ней через интерфейс отладчика можно подключить пакет разработки программного обеспечения. После этого вы можете разрабатывать код RISC-V на языке ассемблера или языках высокого уровня, компилировать его и запускать на процессоре RISC-V, реализованном в ПЛИС, таким же образом, как и на аппаратном процессоре.

Резюме

В этой главе представлены архитектура и набор инструкций процессора RISC-V. Архитектура RISC-V определяет полную спецификацию набора инструкций для пользовательского и привилегированного режимов и ряд расширений для поддержки вычислений общего назначения, высокопроизводительных вычислений и встраиваемых приложений, требующих минимального размера кода. Процессоры RISC-V доступны на рынке. Кроме того, предлагаются бесплатные продукты с открытым исходным кодом для реализации RISC-V на устройствах ПЛИС.

Прочитав эту главу, вы получили знания об архитектуре и возможностях процессора RISC-V и предлагаемых отдельно расширений.

Вы ознакомились с основами набора инструкций RISC-V и теперь понимаете, как его можно адаптировать для поддержки приложений в различных областях — от недорогих маломощных встраиваемых систем до ферм облачных серверов размером с большой склад. Вы также узнали, как реализовать процессор RISC-V на недорогой плате ПЛИС.

В следующей главе вводится концепция виртуализации процессоров, в рамках которой вместо выполнения кода непосредственно на главном процессоре реализуется целая виртуальная среда для запуска одного или нескольких виртуальных процессоров, каждый из которых имеет собственную операционную систему и приложения, на одном физическом процессоре.

Упражнения

1. Посетите сайт **https://www.sifive.com/software/** и скачайте пакет *Freedom Studio*. Это пакет разработки, основанный на **интегрированной среде разработки (IDE)** Eclipse, который оснащен полным набором инструментов для создания приложения RISC-V и его запуска на аппаратном процессоре RISC-V или в среде эмуляции, включенной в комплект Freedom Studio. Установите пакет, следуя указаниям в *руководстве пользователя Freedom Studio*. Запустите Freedom и создайте новый проект Freedom E SDK. В окне создания проекта в качестве цели выберите **qemu-sifive-u54** (это 64-разрядный одноядерный процессор RISC-V в конфигурации RV64GC). Выберите программу-пример `hello` и нажмите кнопку **Finish** (Готово). Запустится сборка программы-примера и эмулятора RISC-V. После завершения сборки отображается окно **Edit Configuration** (Изменение конфигурации). Нажмите **Debug** (Отладка), чтобы запустить программу в среде отладки эмулятора. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается строка "Hello, World!" ("Здравствуй, мир!").
2. Не закрывая проект из *упражнения 1*, в окне **Project** (Проект) найдите файл `hello.c` в папке `src`. Щелкните правой кнопкой мыши на файле и переименуйте его в `hello.s`. Откройте файл `hello.s` в редакторе и удалите все содержимое. Вставьте программу на языке ассемблера, приведенную в *разд. "Язык ассемблера RISC-V" этой главы*. Выполните очистку, а затем снова соберите проект (чтобы запустить операцию очистки, нажмите комбинацию клавиш <Ctrl>+<9>). В меню **Run** (Выполнение) выберите команду **Debug** (Отладка). После запуска отладчика откройте окно для отображения исходного файла `hello.s`, окно **Disassembly** (Дизассемблирование) и окно **Registers** (Регистры). Раскройте дерево **Registers** (Регистры), чтобы отобразить регистры процессора RISC-V. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается строка "Привет, архитектор компьютеров!".
3. Напишите программу на языке ассемблера RISC-V, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: $[(129 - 66) \times (445 + 36)] : 3$. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

12

Виртуализация процессоров

В этой главе представлены концепции, заложенные в основу виртуализации процессоров, и рассматриваются многочисленные преимущества ее эффективного использования для отдельных пользователей и крупных организаций. Мы обсудим основные методы виртуализации, а также открытые и коммерческие инструменты для их реализации.

Средства виртуализации позволяют эмулировать точные представления наборов инструкций различных компьютерных архитектур и операционных систем на компьютерах общего назначения. Виртуализация широко применяется при развертывании реальных программных приложений в облачных средах.

После прочтения этой главы вы будете обладать знаниями о технологиях аппаратной виртуализации и предлагаемых ими преимуществах, а также о том, как современные процессоры поддерживают виртуализацию на уровне набора инструкций. Вы ознакомитесь с техническими особенностями нескольких открытых и коммерческих инструментов, предоставляющих средства виртуализации, и узнаете, как виртуализация используется для создания и развертывания масштабируемых приложений в средах облачных вычислений.

В этой главе будут представлены следующие темы:

- введение в виртуализацию;
- проблемы виртуализации;
- виртуализация современных процессоров;
- инструменты виртуализации;
- виртуализация и облачные вычисления.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Введение в виртуализацию

В области компьютерной архитектуры понятие **"виртуализация"** относится к использованию аппаратных средств и программного обеспечения для создания эмулируемой версии среды для выполнения программ, в отличие от *реальной* среды, в которой обычно выполняется код.

В *главе 7* мы уже рассмотрели одну из форм виртуализации — виртуальную память. Виртуальная память подразумевает использование программного обеспечения в сочетании с поддерживающими аппаратными средствами для создания среды, в которой каждое запущенное приложение функционирует так, как если бы оно имело эксклюзивный доступ ко всем ресурсам компьютера, включая всю требуемую память по предполагаемым адресам. Благодаря этому диапазоны виртуальных адресов, с которыми работает программа, могут совпадать с теми, которыми оперируют другие запущенные в данный момент процессы.

Системы, использующие виртуальную память, создают множество песочниц, в которых каждое приложение выполняется без вмешательства других приложений, за исключением конкуренции за общие системные ресурсы.

В контексте виртуализации **песочница** — это изолированная среда, в которой код выполняется без вмешательства из-за ее пределов, и которая предотвращает воздействие выполняемого внутри нее кода на внешние ресурсы. Однако изоляция между приложениями редко бывает абсолютной. В частности, даже если в системе виртуальной памяти один процесс не может получить доступ к памяти другого процесса, он может сделать что-то еще, например удалить файл, который необходим второму процессу, что может вызвать проблемы для этого процесса.

Основным предметом внимания в этой главе будет виртуализация на уровне процессора, которая позволяет одной или нескольким операционным системам работать на компьютерной системе в виртуализированной среде. Эта виртуальная среда работает на абстрактном уровне относительно физических аппаратных средств системы.

В следующем разделе будут кратко описаны различные типы виртуализации, с которыми вы можете столкнуться.

Типы виртуализации

Термин *"виртуализация"* применяется в нескольких различных вычислительных контекстах, в частности в отношении крупных сетевых сред, таких как коммерче-

ские предприятия, университеты, правительственные организации и поставщики облачных услуг. Приведенные далее определения охватывают наиболее распространенные типы виртуализации, с которыми вы можете встретиться.

Виртуализация операционных систем

Виртуализированная операционная система работает под управлением гипервизора. **Гипервизор** представляет собой комбинацию программного обеспечения и аппаратных средств, которая способна создавать экземпляры виртуальных машин и запускать их. **Виртуальная машина** — это эмуляция целой компьютерной системы. Префикс *гипер-* в слове "гипервизор" указывает на тот факт, что гипервизор имеет более высокий уровень привилегий, чем режим супервизора операционных систем, запущенных на его виртуальных машинах. Другим термином для обозначения гипервизора является **монитор виртуальных машин**.

Различают два основных типа гипервизоров:

- **Гипервизоры типа 1**, иногда называемые *аппаратными*, представляют собой программную среду для управления виртуальными машинами, которая выполняется непосредственно на физическом оборудовании главного компьютера (хоста).
- **Гипервизоры типа 2**, также называемые *хостовыми*, работают как запускаемая в операционной системе хоста прикладная программа, управляющая виртуальными машинами.

ГИПЕРВИЗОР И МОНИТОР ВИРТУАЛЬНЫХ МАШИН



С технической точки зрения, монитор виртуальных машин — это не совсем то же самое, что гипервизор, но для наших целей мы будем рассматривать эти термины как синонимы. Монитор виртуальных машин отвечает за виртуализацию процессора и других компонентов компьютерной системы. А гипервизор объединяет монитор виртуальных машин с базовой операционной системой, которая может быть специально предназначена для размещения виртуальных машин (гипервизор типа 1) или может быть операционной системой общего назначения (гипервизор типа 2).

Компьютер, на котором запущен гипервизор, называют **хостом**. Операционные системы, работающие в управляемых гипервизором виртуальных средах на хосте, называют **гостевыми**.

Независимо от типа, гипервизор позволяет запускать в виртуализированных средах гостевые операционные системы и работающие в них приложения. Он может поддерживать несколько виртуальных машин, одновременно работающих на одном процессоре.

Гипервизор отвечает за управление всеми запросами на привилегированные операции, инициируемые гостевыми операционными системами и запущенными в них приложениями. Каждый запрос требует перехода из пользовательского режима в режим ядра и обратно в пользовательский режим. Как и в невиртуализированных операционных системах, все запросы на ввод-вывод от приложений в гостевых операционных системах связаны с переходом на привилегированный уровень.

Поскольку виртуализация операционной системы с помощью гипервизора типа 2 предполагает запуск операционной системы под гипервизором в операционной системе хоста, возникает естественный вопрос: что произойдет, если вы запустите другую копию гипервизора в операционной системе виртуальной машины? Ответ заключается в том, что такая возможность поддерживается в некоторых, но не во всех комбинациях гипервизора, ОС хоста и гостевой ОС. Такую конфигурацию называют **вложенной виртуализацией**.

Следующим вопросом, который может возникнуть в связи с концепцией вложенной виртуализации: зачем кому-то может это понадобиться? Вот один из возможных сценариев, демонстрирующий пользу вложенной виртуализации: предположим, что присутствие вашего бизнеса в Интернете реализовано в виде виртуализированного образа операционной системы, содержащего различные предустановленные и пользовательские программные компоненты. Если инфраструктура вашего поставщика облачных услуг по какой-либо причине отключится от Интернета, вам потребуется быстро запустить свое приложение у альтернативного поставщика.

Google Compute Engine (**https://cloud.google.com/compute**), например, предоставляет среду выполнения, реализованную в виде виртуальной машины. Compute Engine позволяет установить на эту виртуальную машину гипервизор и запустить в нем виртуальную машину с вашим приложением, оперативно вернув ваше присутствие в Интернете с минимальной потребностью в дополнительных действиях по установке и настройке.

Виртуализация приложений

Вместо создания виртуальной среды для инкапсуляции всей операционной системы возможна виртуализация на уровне отдельного приложения. Виртуализация приложений абстрагирует операционную систему от кода приложения и обеспечивает определенную степень изоляции.

При таком типе виртуализации программы могут выполняться в среде, которая отличается от предусмотренной целевой среды приложений. Например, *Wine* (**https://www.winehq.org/**) — это программа, реализующая слой совместимости приложений, который позволяет программам, написанным для Microsoft Windows, запускаться в операционных системах, совместимых с POSIX, обычно в вариантах ОС Linux.

Интерфейс переносимых операционных систем (portable operating system interface, POSIX) представляет собой набор стандартов IEEE, обеспечивающих совместимость программирования приложений между операционными системами.

Wine реализует эффективную трансляцию обращений к библиотекам и системным функциям Windows в эквивалентные вызовы POSIX.

Виртуализация приложений заменяет части среды выполнения уровнем виртуализации и выполняет такие задачи, как перехват вызовов дискового ввода-вывода и их перенаправление в изолированную виртуализированную дисковую среду.

Виртуализация приложений может инкапсулировать сложный процесс установки программного обеспечения, охватывающий сотни файлов, устанавливаемых в различные каталоги, и многочисленные изменения в реестре Windows, в эквивалентную виртуализированную среду, содержащуюся в одном исполняемом файле.

Простое копирование этого исполняемого файла в целевую систему и его запуск приводят к установке приложения таким же образом, как если бы весь процесс установки происходил в целевой системе.

Виртуализация сетей

Виртуализация сетей — это соединение программных эмуляций сетевых компонентов, таких как коммутаторы, маршрутизаторы, межсетевые экраны и телекоммуникационные сети, таким способом, который отражает физическую конфигурацию этих компонентов. Это позволяет операционным системам и запущенным в них приложениям, взаимодействовать с виртуальной сетью и обмениваться данными по ней таким же образом, как и при физической реализации той же сетевой архитектуры.

Одну физическую сеть можно разделить на несколько **виртуальных локальных сетей** (virtual local area network, VLAN), каждая из которых представляется полнофункциональной изолированной сетью для всех систем, подключенных к ней.

Несколько компьютерных систем в одном и том же физическом местоположении можно подключить к разным VLAN, что фактически равносильно размещению их в отдельных сетях. И наоборот, компьютеры, находящиеся на значительном отдалении друг от друга, можно объединить в одну VLAN, при этом создается впечатление, что они связаны между собой в рамках небольшой локальной сети.

Виртуализация хранилищ

Виртуализация хранилищ — это абстрагирование физического хранилища данных от логической структуры хранения, используемой операционными системами и приложениями. Система виртуализации хранилищ управляет процессом трансляции логических запросов данных в физические операции передачи данных.

Логические запросы данных адресуются как места расположения блоков в разделе диска. После трансляции логических запросов данных в физические запросы операции передачи данных могут в конечном счете работать с устройством хранения, организация которого полностью отличается от логической структуры диска.

Процесс доступа к физическим данным по заданному логическому адресу во многом похож на процесс преобразования виртуального адреса в физический в системах виртуальной памяти. Запрос ввода-вывода логического диска включает в

себя такую информацию, как идентификатор устройства и номер логического блока. Этот запрос преобразуется в идентификатор физического устройства и номер блока. Затем на физическом диске выполняется запрошенная операция чтения или записи.

Виртуализация хранилищ в центрах обработки данных часто включает в себя несколько усовершенствований, повышающих надежность и производительность систем хранения данных. Вот некоторые из них.

- **Централизованное управление** позволяет осуществлять мониторинг и управление большим набором устройств хранения данных, возможно, разной емкости и от разных поставщиков.

  Поскольку все виртуализированные хранилища выглядят для клиентских приложений одинаково, любые различия в устройствах хранения, зависящие от конкретного поставщика, скрыты от пользователей.
- **Репликация** обеспечивает прозрачное резервное копирование данных, предоставляя возможности аварийного восстановления критически важной информации. При выполнении репликации в реальном времени операции записи в массив хранения немедленно копируются в одну или несколько удаленных реплик.
- **Перенос данных** позволяет администраторам перемещать данные в другое физическое местоположение или переключаться на реплику без прерывания параллельных операций ввода-вывода данных. Поскольку система управления виртуализацией хранилища имеет полный контроль над дисковым вводом-выводом, она может в любое время переключить любую логическую операцию чтения или записи с одного физического устройства на другое.

В следующем разделе будут представлены некоторые из наиболее распространенных в настоящее время методов виртуализации процессоров.

Категории виртуализации процессоров

Идеальным режимом работы для среды виртуализации процессоров является **полная виртуализация**. В этом случае двоичный код в операционных системах и приложениях выполняется в виртуальной среде без каких-либо изменений. Для кода гостевой операционной системы, выполняющего привилегированные операции, создается иллюзия, что он имеет полный и единоличный доступ ко всем ресурсам и интерфейсам машины. Гипервизор управляет транзакциями между гостевыми операционными системами и ресурсами хоста и предпринимает любые шаги, необходимые для устранения конфликтов доступа к устройствам ввода-вывода и другим системным ресурсам для каждой виртуальной машины, находящейся под его контролем.

В этой главе мы сосредоточимся на виртуализации процессоров, которая обеспечивает работу полнофункциональных операционных систем и запущенных в них приложений в виртуализированной среде.

Исторически сложилось так, что для реализации виртуализации на уровне процессора использовалось несколько различных подходов. Мы кратко рассмотрим каждый из них, начиная с подхода, впервые реализованного в таких системах, как IBM VM/370, который был представлен в 1972 г. VM/370 была первой операционной системой, разработанной специально для поддержки работы виртуальных машин.

Виртуализация с перехватом и эмуляцией

В вышедшей в 1974 г. статье под названием "Формальные требования к виртуализуемым архитектурам третьего поколения" ("Formal Requirements for Virtualizable Third Generation Architectures") Геральд Попек (Gerald J. Popek) и Роберт П. Голдберг (Robert P. Goldberg) описали три свойства, которыми должен обладать гипервизор для эффективной и полной виртуализации компьютерной системы, включая процессор, память, хранилище и периферийные устройства.

- **Идентичность.** Программы (включая гостевые ОС), работающие под управлением гипервизора, должны демонстрировать поведение, по существу идентичное тому, которое они проявляют при запуске непосредственно на аппаратных средствах компьютера, исключая влияние синхронизации.
- **Управление ресурсами.** Гипервизор должен иметь полный контроль над всеми ресурсами, используемыми виртуальной машиной.
- **Эффективность.** Значительная доля инструкций, выполняемых виртуальной машиной, должна выполняться непосредственно на физическом процессоре без вмешательства гипервизора.

Для того чтобы гипервизор удовлетворял этим критериям, аппаратные средства и ОС компьютера, на котором он запущен, должны предоставить гипервизору полный контроль над виртуальными машинами, которыми он управляет.

Код гостевой ОС предполагает, что он выполняется непосредственно на аппаратных средствах физического процессора и имеет полный контроль над всеми функциями, доступными через аппаратные средства системы. Код гостевой ОС, работающий на уровне привилегий ядра, должен иметь возможность выполнять привилегированные инструкции и получать доступ к областям памяти, зарезервированным для операционной системы.

В гипервизоре, реализующем метод виртуализации с перехватом и эмуляцией, части гипервизора выполняются с привилегиями ядра, в то время как все гостевые ОС (и запущенные в них приложения) работают на уровне привилегий пользователя. Код ядра в гостевых ОС выполняется нормально до тех пор, пока не будет предпринята попытка выполнить привилегированную инструкцию или инструкция доступа к памяти не попробует прочитать или записать память за пределами диапазона адресов пользовательского пространства, доступного гостевой ОС. Когда гостевая ОС попытается выполнить любую из этих операций, произойдет перехват.



ТИПЫ ИСКЛЮЧЕНИЙ: ОШИБКИ, ЛОВУШКИ, АВАРИИ

Термины *"ошибка"*, *"ловушка"* и *"авария"* используются для описания похожих событий исключения. Основные различия между исключениями каждого из этих типов заключаются в следующем.

Ошибка (fault) — это исключение, которое заканчивается перезапуском вызвавшей его инструкции. Например, ошибка страницы (отказ страницы) возникает, когда программа пытается получить доступ к допустимой ячейке памяти, которая в данный момент недоступна. После завершения работы обработчика ошибок страниц вызвавшая это исключение инструкция запускается снова, и выполнение продолжается с этого момента.

Ловушка (trap) — это исключение, которое завершается продолжением выполнения программы, начиная с инструкции, следующей за инструкцией, вызвавшей данное исключение. Например, выполнение возобновляется после исключения, вызванного контрольной точкой отладчика, начиная со следующей инструкции.

Авария (abort) — отражает состояние серьезной ошибки, которая может быть неустранимой. Причинами аварий могут быть такие проблемы, как ошибки доступа к памяти.

Фундаментальный трюк (если можно выразиться таким образом) для внедрения виртуализации с перехватом и эмуляцией заключается в обработке исключений, генерируемых нарушениями привилегий. Во время запуска гипервизор перенаправляет обработчики исключений операционной системы хоста в собственный код. Иными словами, обработчики исключений в гипервизоре получают возможность обрабатывать эти исключения до их обработки в ОС хоста.

Обработчик исключений в гипервизоре проверяет источник каждого исключения, чтобы определить, было ли оно сгенерировано гостевой ОС под управлением гипервизора. Если исключение исходит от гостевой ОС, которой управляет гипервизор, то гипервизор обрабатывает это исключение, эмулируя запрошенную операцию, и возвращает управление выполнением непосредственно гостевой ОС. Если исключение не было порождено гостевой ОС, относящейся к гипервизору, то гипервизор передает это исключение операционной системе хоста для обработки обычным способом.

Для того чтобы виртуализация с перехватом и эмуляцией работала надежно и в полную силу, процессор хоста должен поддерживать критерии, определенные Попеком и Голдбергом. Наиболее важное из этих требований заключается в том, чтобы выполнение любой гостевой инструкции, пытающейся получить доступ к привилегированным ресурсам, генерировало исключение типа "ловушка" (т. е. инициировало перехват). Это необходимо, т. к. система хоста имеет только один набор привилегированных ресурсов (для простоты мы предполагаем, что здесь используется одноядерная система), т. е. ОС хоста и гостевые ОС не могут одновременно осуществлять контроль одних и тех же ресурсов.

В качестве примера привилегированной информации, которой управляет гипервизор, рассмотрим таблицы страниц, используемые для управления виртуальной памятью.

Операционная система хоста поддерживает набор таблиц страниц, которые контролируют всю физическую память системы. Каждая гостевая ОС имеет свой набор таблиц страниц, которые, с ее точки зрения, используются для управления физической и виртуальной памятью в системе, ею управляемой. Эти два набора таблиц страниц содержат существенно различающиеся данные, хотя оба набора в конечном счете взаимодействуют с одними и теми же областями физической памяти.

С помощью механизма ловушек гипервизор перехватывает все попытки гостевой ОС взаимодействовать с таблицами страниц и направляет эти транзакции в область памяти, отведенную для гостевой системы, где содержатся таблицы страниц, используемые только этой гостевой ОС. Затем гипервизор организует необходимую трансляцию адресов, используемых инструкциями, выполняемыми в гостевой ОС, в адреса физической памяти хоста.

Самым большим препятствием для широкого распространения виртуализации в конце 1990-х и начале 2000-х годов был тот факт, что широко используемые в то время процессоры общего назначения (варианты *x*86) не соответствовали критериям виртуализации Попека и Голдберга.

Наборы инструкций *x*86 содержали несколько инструкций, которые позволяли непривилегированному коду взаимодействовать с привилегированными данными без генерирования ловушки. Многие из этих инструкций просто разрешали непривилегированному коду считывать отдельные привилегированные регистры. Такой подход мог показаться безобидным, однако его применение представляло серьезную проблему для виртуализации, поскольку в машине имеется только одна копия каждого из этих регистров, а каждой гостевой ОС может потребоваться держать разные значения в этих регистрах.

В более поздних версиях процессоров семейства *x*86, начиная с 2006 г., были добавлены аппаратные функции **Intel virtualization technology (VT-x)** и **AMD virtualization (AMD-V)**, которые позволили реализовать полную виртуализацию согласно критериям Попека и Голдберга.

Требования к виртуализации, определенные Попеком и Голдбергом, предполагали, что виртуализация с перехватом и эмуляцией, считавшаяся в 1970-х годах единственным практическим методом виртуализации, была единственным осуществимым подходом к виртуализации процессоров. В следующих разделах мы увидим, как можно реализовать действенную и эффективную виртуализацию в компьютерной системе, которая не полностью соответствует критериям Попека и Голдберга.

Паравиртуализация

Поскольку большинство, если не все инструкции, требующие специальной обработки в виртуализированной среде, находятся в гостевой ОС и ее драйверах устройств, одним из способов сделать гостевую среду пригодной для виртуализации является изменение операционной системы и ее драйверов таким образом, чтобы

они явным образом взаимодействовали с гипервизором без использования ловушек. Такой подход называется **паравиртуализацией**.

Этот подход может существенно повысить производительность гостевой ОС по сравнению с системой, работающей под управлением гипервизора с перехватом и эмуляцией, поскольку паравиртуализированный интерфейс гипервизора состоит из оптимизированного кода, а не из серии обращений к обработчикам ловушек. В методе с перехватом и эмуляцией гипервизор должен обрабатывать каждую ловушку в общем обработчике. Этот обработчик прежде всего определяет, является ли источником перехваченного запроса гостевая ОС, которой управляет гипервизор, чтобы определить желаемую операцию и эмулировать ее действие.

Основным недостатком паравиртуализации является необходимость внесения изменений в гостевую операционную систему и ее драйверы для реализации интерфейса с гипервизором. Разработчики дистрибутивов основных операционных систем не проявили особого интереса к полной поддержке интерфейса паравиртуализации.

Двоичная трансляция

Одним из способов обработки проблемных инструкций в процессорных архитектурах, где отсутствует полная поддержка виртуализации, является обнаружение наличия невиртуализуемых инструкций до их выполнения путем сканирования двоичного кода. При обнаружении таких инструкций код преобразуется в инструкции, допускающие виртуализацию и имеющие аналогичное действие.

Такое решение оказалось популярным подходом к виртуализации в архитектуре *x*86. Сочетание метода перехвата и эмуляции с двоичной трансляцией невиртуализируемых инструкций обеспечивает приемлемую производительность гостевой ОС. Этот метод снижает затраты на обработку, необходимую для обращения с невиртуализируемыми инструкциями, до допустимого уровня.

Различают статическую и динамическую двоичную трансляцию. При статической двоичной трансляции набор исполняемых образов перекомпилируется в форму, готовую к выполнению в виртуальной среде. Для выполнения такой трансляции требуется некоторое время, но это однократный процесс, создающий набор системных и пользовательских образов, которые можно использовать до тех пор, пока не будут установлены новые версии образов, что потребует перекомпиляции этих новых образов.

При динамической двоичной трансляции сканирование секций кода для поиска проблемных инструкций осуществляется во время выполнения программы. При обнаружении таких инструкций они заменяются последовательностями виртуализуемых инструкций. Динамическая двоичная трансляция позволяет избежать этапа перекомпиляции, требуемого при статической трансляции, но вызывает снижение производительности из-за непрерывного процесса сканирования и трансляции во время выполнения кода. Каждый сегмент кода необходимо сканировать и транслировать только один раз при каждом запуске программы, после этого он помещается в кеш. Например, код внутри цикла не будет повторно сканироваться на каждой итерации.

Эмуляция аппаратных средств

Все методы виртуализации, которые мы обсуждали до этого момента, предполагали, что гостевая ОС будет работать на процессоре с той же архитектурой набора команд, что и у процессора хоста. Существует множество ситуаций, в которых желательно запускать операционную систему и код приложения на процессоре хоста с совершенно иной архитектурой набора команд (ISA), чем в гостевой ОС.

При эмуляции аппаратных средств процессора каждая инструкция, выполняемая в эмулируемой гостевой системе, должна быть преобразована в эквивалентную инструкцию или последовательность инструкций из ISA хоста. Как и в случае с двоичной трансляцией, для выполнения этого процесса можно использовать статический или динамический подход.

- При **статической трансляции** можно создать эффективный исполняемый образ, способный работать в ISA целевого процессора. Применение статической трансляции сопряжено с некоторым риском, т. к. иногда непросто определить все пути выполнения кода в исполняемом файле, особенно если целевые адреса ветвей вычисляются в коде, а не определены статически. Этот риск также применим к методу статической двоичной трансляции, описанному в предыдущем разделе.
- **Динамическая трансляция** позволяет избежать потенциальных ошибок, которые могут возникнуть при статической трансляции, но производительность при этом может значительно снизиться. Это обусловлено тем, что динамическая трансляция с эмуляцией аппаратных средств включает в себя перевод каждой инструкции из одной архитектуры в другую.

В этом она отличается от динамической двоичной трансляции для той же ISA, при которой также требуется сканировать каждую инструкцию, но трансляция обычно требуется только для небольшой части выполняемых инструкций.

Одним из примеров инструментов эмуляции аппаратных средств является платформа эмуляции и виртуализации с открытым исходным кодом QEMU (**https://www.qemu.org/**). QEMU позволяет с достаточно хорошей производительностью запускать операционные системы для широкого спектра различных процессорных архитектур.

Пакет Freedom Studio для процессора RISC-V включает в себя реализацию QEMU для архитектуры набора инструкций RV64GC. Мы использовали эту виртуализированную среду для выполнения кода в упражнениях *главы 11*.

В следующем разделе мы обсудим проблемы и преимущества, связанные с виртуализацией в процессорных архитектурах, рассмотренных в предыдущих главах.

Проблемы виртуализации

Говоря простыми словами, цель виртуализации процессора состоит в том, чтобы запустить операционную систему в гипервизоре, который либо работает непосред-

ственно на аппаратных средствах компьютерной системы, либо выполняется как приложение под управлением другой операционной системы.

В этом разделе мы сосредоточимся на гипервизорах типа 2 (размещаемых поверх ОС), поскольку этот режим работы создает ряд дополнительных проблем, с которыми аппаратные гипервизоры могут не встретиться, т. к. гипервизоры типа 1 оптимизированы для поддержки виртуализации.

В гипервизорах типа 2 операционная система хоста поддерживает режимы ядра и пользователя, как и гостевая операционная система (с ее точки зрения). Поскольку гостевая ОС и работающие в ней приложения обращаются к системным сервисам, гипервизор должен перехватывать каждый запрос и преобразовывать его в соответствующий вызов операционной системы хоста.

В невиртуализированной системе периферийные устройства, такие как клавиатура и мышь, напрямую взаимодействуют с операционной системой хоста. В виртуализированной среде гипервизор должен управлять интерфейсами с этими устройствами всякий раз, когда пользователь запрашивает взаимодействие с гостевой ОС.

Уровень сложности, связанный с реализацией этих возможностей, зависит от набора команд хоста. Даже если в наборе команд не были реализованы меры для облегчения виртуализации, это не указывает на возможность или невозможность поддержки виртуализации в такой архитектуре прямолинейным способом. Легкость виртуализации в ISA конкретного процессора зависит от того, как процессор обрабатывает небезопасные инструкции.

Небезопасные инструкции

Название метода виртуализации с перехватом и эмуляцией относится к способности гипервизора управлять обработкой исключений, которую обычно выполняют обработчики режима ядра в операционной системе хоста. Это позволяет гипервизору обрабатывать нарушения привилегий и системные вызовы из гостевых операционных систем и приложений, которые выполняются в них.

Каждый раз, когда приложение, запущенное в гостевой ОС, запрашивает системную функцию, например открытие файла, гипервизор перехватывает этот запрос, корректирует его параметры в соответствии с конфигурацией виртуальной машины (возможно, перенаправляя запрос на открытие файла из файловой системы хоста в песочницу виртуального диска гостевой ОС) и передает скорректированный запрос в ОС хоста. Процесс проверки и обработки исключений гипервизором представляет собой фазу эмуляции подхода с перехватом и эмуляцией.

В контексте виртуализации инструкции процессора, которые либо зависят от информации о состоянии привилегированной системы, либо изменяют эту информацию, называются **небезопасными**. Для того чтобы метод перехвата и эмуляции работал безопасно и надежно, все небезопасные инструкции должны генерировать исключения, которые перехватывает гипервизор. Если разрешить выполнение небезопасной инструкции без перехвата, то изоляция виртуальной машины будет нарушена, и работа в режиме виртуализации может завершиться сбоем.

Как мы узнали из *разд. "Виртуализация с перехватом и эмуляцией" этой главы*, данной проблемой были затронуты ранние версии архитектуры *x*86.

Теневые таблицы страниц

Защищенные структуры данных, используемые при распределении и управлении виртуальной и физической памятью, представляют собой еще одну проблему для полной виртуализации. Ядро гостевой ОС предполагает, что оно имеет полный доступ к аппаратным средствам и структурам данных, связанным с системным блоком управления памяти (MMU). Гипервизор должен транслировать запросы гостевой ОС на выделение и освобождение памяти таким способом, который функционально эквивалентен работе гостевой ОС непосредственно на физическом оборудовании.

Особая проблема возникает в архитектуре *x*86, поскольку для правильной настройки системы данные конфигурации таблиц страниц виртуальной памяти должны храниться в процессоре, но эта информация становится недоступной после ее сохранения. Для того чтобы устранить эту проблему, гипервизор поддерживает собственную копию данных конфигурации таблиц страниц, называемых **теневыми таблицами страниц**.

Поскольку теневые таблицы страниц не являются фактическими таблицами страниц, управляющими памятью для ОС хоста, гипервизору необходимо установить ограничения на доступ к областям памяти с теневыми таблицами страниц и перехватывать соответствующие запросы, когда гостевая ОС пытается получить доступ к своим таблицам страниц.

Затем гипервизор эмулирует запрошенную операцию, взаимодействуя с физическим MMU посредством отправления вызовов в ОС хоста.

Использование теневых таблиц страниц приводит к значительному снижению производительности и является одной из приоритетных задач при работе над усовершенствованиями аппаратной виртуализации.

Безопасность

В использовании гипервизора для виртуализации одного или нескольких гостевых приложений нет по существу ничего небезопасного. Однако важно понимать, какие дополнительные возможности открыты для злоумышленников, пытающихся проникнуть в виртуализированную среду.

Гостевая виртуальная машина предоставляет удаленным злоумышленникам, по сути, тот же набор уязвимостей, что и идентичная операционная система и набор приложений, запущенные непосредственно на физическом оборудовании. Гипервизор предоставляет дополнительный путь, который злоумышленник может попытаться использовать в виртуализированной среде.

Если злоумышленникам удастся проникнуть в гипервизор и взять его под контроль, они получат полный доступ ко всем гостевым операционным системам, а также к

приложениям и данным, доступным из этих гостевых ОС. В этом сценарии гостевые ОС открыты для доступа, т. к. они работают на более низком уровне привилегий, нежели гипервизор, который имеет полный контроль над ними.

При внедрении виртуализации в контексте, который допускает публичный доступ, например для веб-хостинга, крайне важно, чтобы учетные данные, позволяющие получить доступ к гипервизорам, были строго ограничены небольшим числом сотрудников. Необходимо строго соблюдать все применимые меры защиты, чтобы предотвратить несанкционированный доступ к гипервизору.

В следующем разделе мы рассмотрим некоторые основные технические аспекты виртуализации, реализованные в современных семействах процессоров.

Виртуализация современных процессоров

Аппаратные архитектуры большинства семейств процессоров общего назначения созрели до такой степени, что они полностью поддерживают выполнение виртуализированных гостевых операционных систем, по крайней мере, в своих наиболее продвинутых версиях. В следующих разделах кратко представлены возможности виртуализации, предоставляемые современными семействами процессоров общего назначения.

Виртуализация процессоров x86

В архитектуре *x*86 изначально не была предусмотрена поддержка выполнения виртуализированных операционных систем.

В результате в процессорах *x*86, начиная с самых ранних моделей и заканчивая серией Pentium, были реализованы наборы команд, содержащие несколько небезопасных и при этом неперехватываемых инструкций. Эти инструкции вызывали проблемы с виртуализацией благодаря тому, например, что позволяли гостевой операционной системе получать доступ к привилегированным регистрам, которые не содержали данных, соответствующих состоянию виртуальной машины.



ТЕКУЩИЙ УРОВЕНЬ ПРИВИЛЕГИЙ И НЕБЕЗОПАСНЫЕ ИНСТРУКЦИИ *x*86

В архитектуре *x*86 два младших бита регистра **сегмента кода** (code segment, CS) содержат значение **текущего уровня привилегий** (current privilege level, CPL), определяющее активное в данный момент кольцо защиты. Значение CPL обычно равно нулю для кода ядра и трем для пользовательских приложений в невиртуализированной операционной системе. В большинстве реализаций гипервизора виртуальные машины выполняются на уровне CPL = 3, что приводит в процессе выполнения к перехвату многих небезопасных инструкций *x*86.

> К сожалению, для первых пользователей виртуализации на *x*86, не все небезопасные инструкции *x*86 в процессорах Pentium вызывали перехват при выполнении на уровне CPL = 3.

Например, инструкция sidt позволяет непривилегированному коду считывать 6-байтное значение **регистра таблицы дескрипторов прерываний** (interrupt descriptor table register, IDTR) и сохранять его в местоположении, указанном в качестве операнда. В одноядерном процессоре *x*86 есть только один регистр IDTR.

Когда гостевая операционная система выполняет эту инструкцию, IDTR содержит данные, связанные с операционной системой хоста и отличающиеся от информации, которую ожидает получить гостевая ОС. Это ведет к ошибочному выполнению гостевой операционной системы.

Запись в IDTR физической системы возможна только для кода, работающего с CPL = 0. Когда гостевая ОС пытается выполнить запись в IDTR во время работы на уровне CPL = 3, происходит нарушение привилегий, и гипервизор обрабатывает последовавшее исключение (ловушку), чтобы эмулировать операцию записи, выполняя вместо этого запись в теневой регистр. Теневой регистр — это просто место в памяти, выделенное гипервизором.

Однако чтение из IDTR на уровне CPL = 3 разрешено. Программное обеспечение пользовательского режима может считывать значение IDTR, и перехват при этом не происходит. Без перехвата гипервизор не может обработать эту операцию чтения и вернуть данные из теневого регистра. Простыми словами, запись в регистр IDTR является виртуализуемой операцией, в чтение из IDTR — нет.

Из сотен инструкций, включенных в архитектуру набора инструкций Pentium, 17 были признаны небезопасными и при этом неперехватываемыми. Другими словами, эти инструкции являются невиртуализируемыми. Таким образом, для архитектуры Pentium *x*86 реализация чистого подхода к виртуализации с перехватом и эмуляцией невозможна.

Небезопасные и неперехватываемые инструкции часто используются в операционных системах и драйверах устройств, но редко встречаются в коде приложений. Гипервизор должен реализовать механизм для обнаружения в коде небезопасных и неперехватываемых инструкций и их обработки.

В нескольких популярных системах виртуализации был принят подход, сочетающий виртуализацию с перехватом и эмуляцией, где это возможно, с двоичной трансляцией небезопасных инструкций в функционально эквивалентные последовательности кода, подходящие для виртуализированной среды.

Большинство приложений для гостевых пользователей вообще не пытаются использовать небезопасные инструкции. Это позволяет им работать на полной скорости, после того как гипервизор просканирует код, чтобы убедиться в отсутствии небезопасных инструкций. Однако код гостевого ядра может содержать многочисленные, часто встречающиеся небезопасные инструкции. Для достижения прием-

лемой производительности кода при использовании двоичной трансляции необходимо кешировать измененный код при его первом выполнении и повторно использовать кешированную версию при последующих проходах выполнения.

Аппаратная виртуализация процессоров *x*86

В 2005 и 2006 гг. Intel и AMD выпустили версии процессоров *x*86, содержащие аппаратные расширения, поддерживающие виртуализацию.

Эти расширения разрешили проблемы, вызванные небезопасными и неперехватываемыми инструкциями, обеспечив возможность полной виртуализации систем в соответствии с критериями Попека и Голдберга. Эти расширения получили названия AMD-V в процессорах AMD и VT-x в процессорах Intel. Расширения виртуализации в современных процессорах Intel носят название VT.

Первоначальные реализации этих технологий аппаратной виртуализации устранили требования к двоичной трансляции небезопасных инструкций, но после исключения двоичной трансляции общая производительность виртуальных машин существенно не улучшилась. Это было связано с тем, что по-прежнему требовалось использовать теневые таблицы страниц, на долю которых приходилась большая часть снижения производительности, наблюдаемого во время работы виртуальных машин.

В более поздних версиях технологии аппаратной виртуализации были устранены многие факторы снижения производительности при работе виртуальных машин, что привело к широкому внедрению виртуализации семейства *x*86 в различных областях. Сегодня доступно множество инструментов и платформ для реализации решений виртуализации *x*86 различного масштаба — от автономной рабочей станции до полностью управляемого центра обработки данных с поддержкой тысяч серверов, каждый из которых способен одновременно поддерживать работу нескольких виртуальных машин.

Виртуализация процессоров ARM

Архитектура ARM поддерживает виртуализацию как в 32-разрядном, так и в 64-разрядном режиме. Аппаратная поддержка виртуализации в этой архитектуре включает в себя следующие компоненты:

- полнофункциональную виртуализацию с перехватом и эмуляцией;
- выделенную категорию исключений для использования гипервизором;
- дополнительные регистры, поддерживающие исключения гипервизора и указатели стека.

Архитектура ARM обеспечивает аппаратную поддержку трансляции запросов на доступ к памяти из гостевых операционных систем в адреса физической системы.

Системы, работающие на процессорах ARM, предлагают широкие возможности для выполнения виртуальных машин с использованием гипервизора типа 1 или 2. Производительность 64-разрядного процессора ARM сопоставима с серверами *x*64 с аналогичными характеристиками. Для многих областей применения, таких как

развертывание крупных центров обработки данных, выбор между *x*64 и ARM в качестве серверного процессора может зависеть от факторов, не связанных с производительностью процессора, таких как энергопотребление или требования к охлаждению.

Виртуализация процессоров RISC-V

В отличие от других архитектур набора инструкций (ISA), рассмотренных в этой главе, в ISA RISC-V всесторонняя поддержка виртуализации была базовым требованием с самого начала процесса проектирования. Расширение RISC-V для реализации гипервизора предоставляет полный набор возможностей для поддержки гипервизоров обоих типов.

В этом расширении RISC-V полностью реализован метод виртуализации с перехватом и эмуляцией и обеспечена аппаратная поддержка трансляции физических адресов гостевой операционной системы в физические адреса хоста.

В RISC-V реализована концепция приоритетного и фонового управления и регистров состояния, которая позволяет быстро переключать регистры супервизора в рабочее состояние и из него по мере перехода виртуальных машин в состояние выполнения и из него.

Каждый аппаратный поток в RISC-V выполняется на одном из трех уровней привилегий:

- **пользователь (U)** — то же самое, что и привилегии пользователя в традиционной операционной системе;
- **супервизор (S)** — то же самое, что и режим супервизора или ядра в традиционной операционной системе;
- **машина (M)** — наивысший уровень привилегий с доступом ко всем функциям системы.

Некоторые процессоры могут реализовывать все три режима, другие — лишь два из них, M и U, третьи — только режим M. Иные комбинации не допускаются.

В процессоре RISC-V, поддерживающем расширение гипервизора, режимом виртуализации управляет дополнительный бит конфигурации — **бит V**. Для аппаратных потоков, выполняющихся в виртуализированной гостевой системе, бит V установлен в состояние 1. Если бит V имеет значение 1, доступны уровни привилегий как пользователя, так и супервизора. Эти уровни называют режимом **виртуального пользователя** (virtual user, VU) и режимом **виртуального супервизора** (virtual supervisor, VS).

В контексте гипервизора RISC-V режим супервизора с битом V, равным 0, переименован в режим **гипервизора — расширенного супервизора** (Hypervisor-extended Supervisor mode, HS). Это название указывает на то, что HS — это режим, в котором работает сам гипервизор, независимо от его типа (1 или 2). Оставшийся уровень привилегий M действует только в невиртуализированном режиме при V = 0.

В режимах VU и VS процессор RISC-V реализует двухуровневую схему трансляции адресов для преобразования каждого виртуального адреса гостевой ОС сначала в ее физический адрес, а затем в физический адрес супервизора. Эта процедура обеспечивает эффективную трансляцию виртуальных адресов в приложениях, запущенных в гостевых операционных системах, в физические адреса в системной памяти.

В следующем разделе представлены обзоры нескольких популярных инструментов для виртуализации процессоров и операционных систем.

Инструменты виртуализации

В этом разделе мы рассмотрим несколько широко доступных инструментов с открытым исходным кодом и коммерческих инструментов, которые реализуют различные формы виртуализации процессоров. Эта информация может быть полезна в качестве отправной точки в начале работы над проектом, связанным с виртуализацией.

VirtualBox

VirtualBox — это бесплатный гипервизор типа 2 с открытым исходным кодом от корпорации Oracle. Поддерживаемые операционные системы хоста — Windows и несколько дистрибутивов Linux. Одна или несколько гостевых операционных систем на одном хосте могут одновременно запускать Windows, macOS, Solaris, Open Solaris и различные дистрибутивы Linux.

ТРЕБОВАНИЯ К ЛИЦЕНЗИРОВАНИЮ ГОСТЕВЫХ ОС



Для того чтобы организации и частные лица соблюдали законы об авторском праве, операционные системы, требующие лицензирования, такие как Windows, должны иметь действующие лицензии даже при запуске в качестве гостевых операционных систем.

Отдельные виртуальные машины можно запускать, останавливать и приостанавливать под управлением интерактивной программы управления VirtualBox или из командной строки. VirtualBox позволяет делать моментальные снимки выполняющихся виртуальных машин и сохранять их на диск. Впоследствии с помощью сохраненного моментального снимка можно возобновить выполнение машины с той точки, в которой был сделан этот снимок.

VirtualBox требует аппаратной виртуализации, предоставляемой платформами с расширениями AMD-V или Intel VT. Предусмотрено множество механизмов, с помощью которых виртуальные машины могут взаимодействовать с ОС хоста и друг с другом.

Общий буфер обмена поддерживает операции копирования-вставки между хостом и гостевой машиной, а также между гостевыми машинами. В VirtualBox можно настроить внутреннюю сеть, чтобы гостевые ОС могли взаимодействовать друг с другом так, как если бы они были подключены к изолированной локальной сети.

VMware Workstation

ПО VMware Workstation, первая версия которого была выпущена в 1999 г., — это гипервизор типа 2, который работает на 64-разрядных версиях Windows и Linux. Продукты VMware предлагаются на коммерческой основе и требуют приобретения лицензий некоторыми пользователями.

Версия VMware Workstation под названием VMware Workstation Player доступна бесплатно с условием, что она будет использоваться только в некоммерческих целях.

VMware Workstation поддерживает выполнение (возможно) нескольких копий операционных систем Windows, Linux и MS-DOS в операционной системе Linux или Windows хоста. Как и VirtualBox, VMware Workstation может создавать моментальные снимки состояния виртуальной машины, сохранять эту информацию на диск и позже возобновлять выполнение, начиная с момента создания снимка. VMware Workstation также поддерживает функции связи между хостом и гостевой системой и между самими гостевыми системами, такие как общий буфер обмена и эмуляция локальной сети.

VMware ESXi

ESXi — это гипервизор типа 1, предназначенный для развертывания систем корпоративного класса в центрах обработки данных и фермах облачных серверов. Как гипервизор типа 1, ESXi работает на физическом оборудовании хоста. Он имеет интерфейсы с аппаратными средствами компьютерной системы, каждой гостевой операционной системой и интерфейсом управления, называемым консолью обслуживания.

С консоли обслуживания администраторы могут осуществлять контроль и управление работой крупномасштабного центра обработки данных, вызывая виртуальные машины и назначая им задачи (называемые **рабочими нагрузками**).

ESXi предоставляет дополнительные функции, необходимые для крупномасштабных развертываний, такие как мониторинг производительности и обнаружение неисправностей. В случае аппаратного сбоя или для обслуживания системы рабочие нагрузки виртуальных машин можно легко перенести на другие хосты.

KVM

Kernel-based Virtual Machine (KVM) — это гипервизор типа 2 с открытым исходным кодом, первая версия которого была выпущена в 2007 г. KVM поддерживает полную виртуализацию для гостевых операционных систем. При использовании с

хостами *x*86 или *x*64 системные аппаратные средства должны включать расширение виртуализации AMD-V или Intel VT. Ядро гипервизора KVM включено в основную линейку разработки Linux.

KVM поддерживает выполнение одного или нескольких виртуализированных экземпляров Linux и Windows на хосте без какой-либо модификации гостевых операционных систем.

Изначально KVM был разработан для 32-разрядной архитектуры *x*86, однако впоследствии он был перенесен на *x*64, ARM и PowerPC. KVM поддерживает паравиртуализацию для гостевых систем Linux и Windows с использованием Virtio API. В этом режиме данное решение предоставляет паравиртуализированные драйверы устройств для Ethernet, дискового ввода-вывода и графического дисплея.

Xen

Первая версия Xen, бесплатного гипервизора типа 1 с открытым исходным кодом, была выпущена в 2003 г. Текущая версия Xen работает на процессорах *x*86, *x*64 и ARM. Xen поддерживает гостевые виртуальные машины, работающие под управлением аппаратной виртуализации (AMD-V или Intel VT) или в качестве паравиртуализированных операционных систем. Xen реализован в основной ветви ядра Linux.

Гипервизор Xen запускает одну виртуальную машину на самом привилегированном уровне, называемом доменом 0 или dom0. Виртуальная машина dom0 обычно является вариантом Linux и имеет полный доступ к системным аппаратным средствам. Машина dom0 предоставляет пользовательский интерфейс для управления гипервизором.

Некоторые из крупнейших коммерческих поставщиков облачных услуг, включая Amazon EC2, IBM SoftLayer и Rackspace Cloud, используют Xen в качестве своей основной гипервизорной платформы.

Xen поддерживает функцию динамического переноса, благодаря которой виртуальную машину можно перенести с одной хост-платформы на другую без остановки ее работы.

QEMU

QEMU (от англ. Quick EMUlator — быстрый эмулятор) — это бесплатный эмулятор с открытым исходным кодом, который реализует аппаратную виртуализацию. QEMU может выполнять эмуляцию на уровне отдельного приложения или всей компьютерной системы. На уровне приложений QEMU может запускать отдельные приложения Linux или macOS, которые были созданы для архитектуры ISA, отличной от ISA среды выполнения.

При эмуляции системы QEMU представляет собой законченную компьютерную систему, включая периферийные устройства.

QEMU поддерживает одновременное выполнение нескольких гостевых операционных систем на одном хосте. Поддерживаемые ISA: *x*86, MIPS, ARMv7, ARMv8, PowerPC, Sparc, Alpha, OpenRISC и RISC-V.

QEMU поддерживает настройку и перенос машин KVM, выполняя аппаратную эмуляцию совместно с виртуальной машиной, работающей под управлением KVM. Аналогичным образом QEMU может обеспечить аппаратную эмуляцию для виртуальных машин, работающих под управлением Xen.

Среди прочих инструментов виртуализации QEMU уникален тем, что ему не требуется запуск с повышенными привилегиями, поскольку он полностью эмулирует гостевую систему в программном обеспечении. Недостатком такого подхода является снижение производительности, обусловленное процессом эмуляции программного обеспечения.

В следующем разделе мы обсудим синергетические эффекты, возникающие в результате использования виртуализации в среде облачных вычислений.

Виртуализация и облачные вычисления

Термины *"виртуализация"* и *"облачные вычисления"* часто употребляют с размытыми, иногда перекрывающими друг друга значениями. Разницу между ними можно определить следующим образом:

- виртуализация — это технология абстрагирования программных систем от среды, в которой они работают;
- облачные вычисления — это методология использования виртуализации и других технологий для обеспечения развертывания, мониторинга и управления крупными центрами обработки данных.

Использование виртуализации в средах облачных вычислений позволяет гибко развертывать рабочие нагрузки приложений в крупных парках универсального вычислительного оборудования контролируемым и согласованным образом. Размещая такие приложения, как веб-серверы, в виртуальных машинах, можно динамически масштабировать доступные вычислительные мощности в соответствии с изменяющимися условиями загрузки.

Коммерческие поставщики облачных услуг, как правило, предлагают использовать свои системы с оплатой за фактическое использование ресурсов. Посещаемость веб-сайта с обычно небольшим объемом трафика может резко вырасти, если, например, он будет упомянут в национальной программе новостей. Если сайт развернут в масштабируемой облачной среде, управляющее ПО обнаружит увеличение нагрузки и запустит дополнительные экземпляры этого веб-сайта и, возможно, серверной базы данных.

Такое повышенное потребление ресурсов приведет к увеличению счета, выставляемого поставщиком облачных услуг, который большинство компаний с радостью оплатят, если в результате их веб-сайт сохранит работоспособность и будет нормально реагировать на ввод данных пользователем даже при значительной нагрузке.

Среды управления облачными ресурсами, такие как VMware ESXi и Xen, предоставляют комплексные инструменты для настройки, развертывания, управления и обслуживания крупномасштабных облачных операций. Такие конфигурации могут быть предназначены для локального использования отдельной организацией или могут предлагать общедоступные ресурсы для поставщиков онлайн-услуг, таких как Amazon Web Services.

Потребление электроэнергии

Для поставщиков облачных услуг плата за потребленную электроэнергию относится к значительным расходам. Каждый включенный компьютер в крупной серверной ферме потребляет электроэнергию, даже если он не выполняет никакой полезной работы. Для инфраструктуры, содержащей тысячи компьютеров, важно, чтобы серверы потребляли электроэнергию только тогда, когда это необходимо оплачивающим их услуги клиентам.

Виртуализация существенно помогает в эффективном использовании серверных систем.

Поскольку на одном сервере можно разместить несколько гостевых виртуальных машин, рабочие нагрузки клиентов могут быть эффективно распределены по серверному оборудованию таким образом, чтобы избежать низкой загрузки доступных компьютеров. Серверы, которые не нужны в данный момент, можно выключить, чтобы снизить потребление электроэнергии, что, в свою очередь, уменьшает затраты поставщика облачных услуг и обеспечивает более конкурентоспособные цены для конечных пользователей.

В этом разделе представлено лишь краткое введение в использование виртуализации в контексте облачных вычислений. Большинство организаций и частных лиц, имеющих представительство в Интернете, используют виртуальные серверы в среде облачных вычислений, независимо от того, знают они об этом или нет.

Резюме

В этой главе были представлены концепции, заложенные в основу виртуализации процессоров, и объяснены многочисленные преимущества ее эффективного использования для отдельных пользователей и крупных организаций. Мы рассмотрели основные методы виртуализации, а также инструменты, которые их реализуют, — как коммерческие, так и с открытым исходным кодом.

Мы также ознакомились с преимуществами виртуализации при развертывании реальных программных приложений в облачных средах.

Теперь вы должны хорошо разбираться в технологиях виртуализации процессоров и предлагаемых ими преимуществах, а также в том, как ISA современных процессоров поддерживают виртуализацию на уровне набора инструкций. Мы рассмотрели несколько инструментов с открытым исходным кодом и коммерческих инструментов, предоставляющих возможности виртуализации. Теперь вы понимаете, как

можно использовать виртуализацию для создания и развертывания масштабируемых приложений в средах облачных вычислений.

В следующей главе мы рассмотрим архитектуру решений для нескольких категорий применения, включая мобильные устройства, персональные компьютеры, игровые системы, системы для обработки больших данных и нейронные сети.

Упражнения

1. Скачайте и установите текущую версию VirtualBox. Скачайте, установите и запустите Ubuntu Linux в качестве виртуальной машины в VirtualBox. Подключите гостевую ОС к Интернету с помощью мостового сетевого адаптера. Настройте и включите общий доступ к буферу обмена и файлам между гостевой системой Ubuntu и операционной системой хоста.
2. В операционной системе Ubuntu, которую вы установили в *упражнении 1*, установите VirtualBox, затем установите и запустите версию виртуальной машины FreeDOS, которую можно найти по адресу **https://www.freedos.org/download/**.

   Убедитесь в правильности выполнения команд DOS, таких как `echo Hello World!` и `mem`, в виртуальной машине FreeDOS. Выполнение этого упражнения означает, что вы реализовали экземпляр вложенной виртуализации.
3. Создайте две отдельные копии гостевой машины Ubuntu в среде хоста VirtualBox. Настройте обе гостевые системы Ubuntu для подключения к внутренней сети VirtualBox. Задайте в этих двух машинах совместимые IP-адреса. Убедитесь, что каждая машина может получить ответ от другой, используя команду `ping`. Выполнение этого упражнения означает, что вы настроили в своей виртуализированной среде виртуальную сеть.

13

Специализированные компьютерные архитектуры

В этой главе собраны воедино темы, обсуждавшиеся в предыдущих главах, поскольку здесь мы рассмотрим архитектуры различных компьютерных систем, разработанных для реализации уникальных требований пользователей. Мы разберемся в требованиях на уровне пользователя и функциональных возможностях, относящихся к нескольким категориям реальных компьютерных систем.

В этой главе будут рассмотрены следующие темы:

- проектирование архитектуры компьютерных систем на основе уникальных требований;
- архитектура смартфона;
- архитектура персонального компьютера;
- вычислительная архитектура масштаба ЦОД (центра обработки данных);
- архитектура процессоров для нейронных сетей и машинного обучения.

Эта глава поможет вам разобраться в процессе принятия решений при определении компьютерных архитектур, которые должны обеспечить поддержку конкретных потребностей. Вы ознакомитесь с ключевыми требованиями, определяющими архитектуру мобильных устройств, персональных компьютеров, облачных серверных систем, нейронных сетей и других средств машинного обучения.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Проектирование архитектуры компьютерных систем на основе уникальных требований

Каждое устройство, содержащее цифровой процессор, предназначено для выполнения определенной функции или набора функций. К подобным устройствам относятся устройства общего назначения, такие как персональные компьютеры. Исходную информацию для начала проектирования архитектуры цифровых компонентов конкретного устройства или компьютерной системы содержит общий перечень всех необходимых и желаемых функций и возможностей.

Некоторые соображения, которые разработчик архитектуры компьютеров должен учитывать в процессе проектирования цифровой системы, приведены в следующем списке.

- **Типы требуемой обработки.** Должно ли устройство обрабатывать аудио, видео или другую аналоговую информацию? Включен ли в проект графический дисплей высокого разрешения? Потребуется ли обширное применение арифметики с плавающей запятой или с целыми числами? Должна ли система поддерживать работу нескольких одновременно запущенных приложений? Будут ли использоваться специальные алгоритмы, такие как обработка с помощью нейронных сетей?
- **Требования к памяти и хранилищу.** Какой объем оперативной памяти потребуется операционной системе и предполагаемым пользовательским приложениям для надлежащей работы? Какой объем энергонезависимого хранилища потребуется?
- **Система жесткого или мягкого реального времени.** Является ли обязательным реагирование на входные данные в реальном времени в течение определенного срока? Если работа в реальном времени не является абсолютным требованием, существует ли желаемое время отклика, которое должно соблюдаться большую часть времени (но не постоянно)?
- **Требования к подключению.** Какие типы проводных подключений, таких как Ethernet и USB, должно поддерживать устройство? Сколько физических портов требуется для подключения каждого типа? Какие типы беспроводных подключений (сотовая сеть, Wi-Fi, Bluetooth, NFC, GPS и т. д.) необходимы?
- **Энергопотребление.** Использует ли устройство аккумуляторное питание? Если да, то каков допустимый уровень энергопотребления компонентов цифровой системы в периоды ее интенсивного использования, а также в периоды простоя? Если система работает от внешнего источника питания, что для нее важнее — высокая производительность обработки или низкое энергопотребление? Для систем с аккумуляторным питанием и систем с внешним питанием — каковы пределы рассеивания мощности, после превышения которых перегрев становится проблемой?

- **Физические ограничения.** Существуют ли жесткие ограничения на размер компонентов цифровой обработки? Существует ли ограничение по массе системы?
- **Ограничения по условиям окружающей среды.** Должно ли устройство быть рассчитано на работу при крайне высоких или низких температурах? Какой уровень вибрации и ударных воздействий должно выдерживать устройство? Должно ли устройство работать в условиях повышенной влажности, сухости или запыленности атмосферы? Будет ли устройство подвергаться воздействию соленой воды или радиации в космосе?

В следующих разделах рассматриваются общие архитектуры цифровых устройств нескольких категорий и обсуждаются ответы, к которым пришли архитекторы этих систем, отвечая на вопросы, подобные тем, что приведены выше. Мы начнем с архитектуры мобильных устройств, обратив особое внимание на iPhone 13 Pro Max.

Архитектура смартфона

Для того чтобы получить широкое признание, смартфон должен обладать тремя ключевыми характеристиками на уровне архитектуры: небольшие размеры (за исключением дисплея), длительное время автономной работы и очень высокая производительность обработки по требованию. Очевидно, что требования длительного времени автономной работы и высокой вычислительной мощности противоречат друг другу и должны быть сбалансированы для получения оптимального решения.

Требование небольших размеров обычно удовлетворяют, начиная с оценки размера экрана (по высоте и ширине), который должен быть достаточно большим для отображения высококачественного видео и выполнения функций устройства пользовательского ввода (особенно в качестве клавиатуры), но при этом устройство должно быть достаточно компактным, чтобы е о можно было легко носить в кармане или сумочке. Для того чтобы сохранить небольшой размер устройства с точки зрения общего объема, оно должно быть как можно более тонким.

В стремлении к тонкому профилю конструкция механической части должна обеспечивать достаточную прочность, чтобы удерживать экран и противостоять повреждениям при обычном обращении, падениях на пол и других физических воздействиях, одновременно оставляя достаточно места для аккумулятора, цифровых компонентов и подсистем, таких как приемопередатчик сотовой связи.

Поскольку пользователи будут иметь неограниченный доступ к внешним и внутренним компонентам своих телефонов, любые коммерческие секреты или другая интеллектуальная собственность, такие как встроенное программное обеспечение системы, которые изготовитель желает сохранить в тайне, должны быть защищены от всех возможных способов извлечения. Тем не менее даже при наличии подобных средств защиты конечным пользователям должна быть обеспечена возможность простой и безопасной установки обновлений встроенного ПО с одновременной блокировкой установки неавторизованных образов встроенного ПО.

В следующем разделе мы рассмотрим цифровую архитектуру современного смартфона высокого класса в контексте этих требований.

iPhone 13 Pro Max

iPhone 13 Pro Max был выпущен в сентябре 2021 г. На момент своего выпуска он был флагманским смартфоном Apple и совмещал в себе ряд самых передовых технологий на рынке.



Поскольку Apple публикует лишь ограниченную информацию о деталях устройства своих изделий, некоторые из приведенных ниже сведений получены в результате разборки и других видов анализа обозревателями iPhone 13 Pro Max, и поэтому к ним следует отнестись с определенной долей скептицизма.

В основе вычислительной архитектуры iPhone 13 Pro Max — система на кристалле (SoC) Apple A15 Bionic с шестиядерным процессором ARMv8, состоящим из 15 млрд КМОП-транзисторов. Два ядра с архитектурой под кодовым названием **Avalanche** оптимизированы для работы с высокой производительностью и поддерживают максимальную тактовую частоту 3,23 ГГц. Остальные четыре ядра под кодовым названием **Blizzard** предназначены для экономичной работы на частоте до 1,82 ГГц. Все шесть ядер представляют собой суперскалярные конструкции с внеочередным выполнением инструкций. При одновременном выполнении нескольких процессов или нескольких потоков в рамках одного процесса возможна параллельная работа всех шести ядер.

Конечно, одновременная работа шести ядер приводит к значительному расходу энергии. Бóльшую часть времени, особенно когда пользователь не взаимодействует с устройством, несколько ядер переводятся в режимы с низким энергопотреблением, чтобы максимально увеличить время автономной работы.

iPhone 13 Pro Max содержит до 8 Гбайт оперативной памяти четвертого поколения **с пониженным энергопотреблением и удвоенной скоростью передачи данных** (low-power double data rate RAM, LP-DDR4x). Каждое устройство LP-DDR4x поддерживает скорость передачи данных 4266 Мбит/с. Усовершенствование, на которое указывает буква *x* в названии LP-DDR4x, уменьшает напряжение сигнала ввода-вывода с 1,1 В в предыдущем поколении DDR (LP-DDR4) до 0,6 В в LP-DDR4x, снижая энергопотребление оперативной памяти.

В систему на кристалле A15 интегрирован пятиядерный графический процессор (GPU), разработанный Apple. В дополнение к ускорению решения традиционных задач GPU, таких как визуализация трехмерных сцен, в этом графическом процессоре реализовано несколько улучшений, обеспечивающих поддержку машинного обучения и других задач параллельной обработки данных, подходящих для реализации на аппаратных средствах GPU.

Процесс 3D-визуализации основан на алгоритме **отложенного рендеринга на основе плиток** (tile-based deferred rendering, TBDR), адаптированном для систем с ограниченными ресурсами, таких как смартфоны. В процессе рендеринга алгоритм TBDR пытается как можно раньше идентифицировать невидимые в поле зрения объекты (т. е. те, которые заслонены другими объектами), чтобы исключить работу по их отрисовке. Этот процесс разделяет изображение на элементы (плитки) и для достижения максимальной производительности выполняет алгоритм TBDR для нескольких плиток параллельно.

A15 содержит нейронный процессор под названием **Apple Neural Engine**. Он имеет 16 ядер, способных выполнять в общей сложности 15,8 трлн операций в секунду. Данная подсистема, по-видимому, используется для таких задач, как выявление и отслеживание объектов в видеопотоке, поступающем в реальном времени с камер телефона.

A15 содержит сопроцессор движения. Это отдельный процессор ARM, предназначенный для сбора и обработки данных, поступающих от гироскопа, акселерометра, компаса и барометрических датчиков телефона. Результат обработки этих данных представляет собой оценочную категорию текущей активности пользователя, такую как ходьба, бег, сон или вождение автомобиля. Сбор и обработка данных с датчиков продолжается и при низком уровне энергопотребления, когда остальная часть телефона находится в спящем режиме.

Система A15, полностью соответствующая термину *"система на кристалле"*, также содержит высокопроизводительный контроллер **твердотельного накопителя** (solid-state drive, SSD), который управляет доступом к внутреннему хранилищу объемом до 1 Тбайт. В качестве интерфейса между A15 и флеш-памятью используется PCI Express.

На рис. 13.1 показаны основные компоненты iPhone 13 Pro Max.

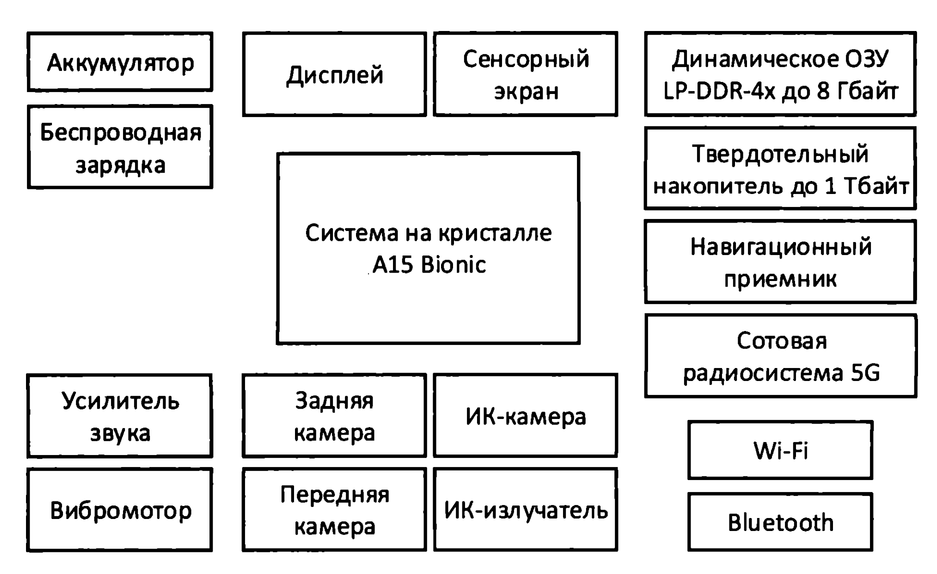


Рис. 13.1. *Компоненты iPhone 13 Pro Max*

iPhone 13 Pro Max содержит несколько высокоэффективных подсистем, описанных в табл. 13.1.

Таблица 13.1. *Подсистемы iPhone 13 Pro Max*

Подсистема	Описание
Аккумулятор	iPhone 13 Pro Max содержит аккумулятор емкостью 3095 **миллиампер-часов** (мА·ч)
Дисплей	Плоскопанельный дисплей с диагональю 6,1 дюйма (155 мм) и разрешением 2532 × 1170 пикселов. Дисплей изготовлен по технологии, основанной на **органических светодиодах** (organic light-emitting diode, OLED), где слово *"органический"* указывает на использование органических соединений в люминесцентном материале
Сенсорный экран	Для обнаружения прикосновений в дисплей встроены емкостные датчики. Они обнаруживают изменение емкости, возникающее в результате приближения к ним проводящего объекта, такого как человеческий палец. После фильтрации и обработки исходных измерений датчиков можно определить точные местоположения нескольких точек одновременных касаний экрана. Кроме того, датчики измеряют давление, прилагаемое при прикосновении к экрану. Это позволяет программам по-разному реагировать на жесткие и мягкие нажатия на экран
Несколько камер, ИК-излучатель, ИК-камера	Каждая из трех задних камер создает изображения размером 12 **мегапикселов** (**Мп**). Одна камера имеет стандартный объектив, вторая — телеобъектив, третья — сверхширокоугольный объектив. Эти камеры способны записывать видео в формате 4K (3840 × 2160 пикселов) с частотой до 60 **кадров в секунду** (**кадр/с**) или 1080p (1920 × 1080 пикселов) с частотой до 240 кадр/с. Передняя камера разрешением 12 Мп записывает видео в формате 4K с частотой 60 кадр/с. Передняя панель iPhone 13 Pro Max содержит отдельный **инфракрасный** (**ИК**) лидарный датчик, поддерживающий распознавание лиц. Эта функция использует ИК-излучатель для подсветки 30 000 точек, которые генерируют трехмерную карту лица пользователя. Телефон использует эту карту для проверки личности пользователя и снятия блокировки при подтверждении соответствия
Беспроводная зарядка	iPhone 13 Pro Max поддерживает функции беспроводной зарядки **Apple MagSafe** мощностью до 15 Вт и **Qi** мощностью до 7,5 Вт
Навигационные приемники	iPhone 13 Pro Max оснащен приемниками сигналов спутниковых навигационных систем **Global Positioning System** (GPS), **ГЛОНАСС**, **Galileo**, **Quasi-Zenith Satellite System** (QZSS) и **BeiDou**
Радиосистема сотовой связи	iPhone 13 Pro Max содержит модем сотовой радиосвязи **пятого поколения (5G)**

Таблица 13.1 *(окончание)*

Подсистема	Описание
Wi-Fi и Bluetooth	iPhone 13 Pro Max содержит интерфейс Wi-Fi, поддерживающий стандарт Wi-Fi 6 (802.11ax) с технологией 2x2 MIMO. Технология 2x2 MIMO использует две передающие и две приемные антенны для исключения пропадания сигнала. Интерфейс Bluetooth поддерживает версию 5.0 стандарта Bluetooth
Усилитель звука, вибромотор	Усилитель звука в iPhone 13 Pro Max потребляет крайне мало энергии в режиме ожидания и обеспечивает высокую эффективность и превосходное качество звука во время работы. Вибрация создается устройством **тактильного отклика** — линейным вибратором, способным генерировать различные тактильные сигналы обратной связи с пользователем

iPhone 13 Pro Max объединил в себе самые передовые, компактные и легкие мобильные электронные технологии, доступные на момент его разработки, с изящным и привлекательным корпусом, став флагманским продуктом линейки смартфонов Apple.

Далее мы рассмотрим архитектуру высокопроизводительного персонального компьютера.

Архитектура персонального компьютера

Следующая система, которую мы рассмотрим, — это игровой ПК с процессором, который на момент написания статьи (в конце 2021 г.) был лидером по производительности. Мы подробно рассмотрим процессор системы, графический процессор и основные подсистемы компьютера.

Игровой настольный компьютер Alienware Aurora Ryzen Edition R10

Настольный ПК Alienware Aurora Ryzen Edition R10 был разработан с целью обеспечить максимальную производительность для игровых приложений. Для достижения высокой скорости архитектура этой системы построена на основе самых быстрых компонентов — центрального процессора, графического процессора, подсистем оперативной и дисковой памяти, предлагаемых по цене, которую могут позволить себе как минимум некоторые серьезные геймеры и другие пользователи, ценящие высокую производительность. Однако количество покупателей, интересующихся высокопроизводительными элементами конфигурации, описанными в этом разделе, вероятно, будет ограничено их стоимостью, которая составляет более 4000 долларов США.

Aurora Ryzen Edition R10 комплектуется различными процессорами AMD Ryzen с разным уровнем производительности и ценой. В настоящее время самым производительным процессором для этой платформы является AMD Ryzen 9 5950X, выпущенный в ноябре 2020 г.

Ryzen 9 5950X реализует 64-разрядную ISA в суперскалярной архитектуре с внеочередным выполнением инструкций, упреждением и переименованием регистров. Согласно предоставленным AMD данным, микроархитектура Zen 3 процессора 5950X имеет показатель производительности, выраженный в количестве **инструкций за такт процессора** (instructions per clock, IPC) на 19% выше, чем у микроархитектуры AMD предыдущего поколения (Zen 2).

Процессор Ryzen 9 5950X имеет следующие особенности:

- 16 ядер;
- 2 потока на процессор (в общей сложности 32 одновременных потока);
- базовая тактовая частота 3,4 ГГц с пиковой частотой 4,9 ГГц при разгоне;
- кеш-память инструкций уровня L1 на 32 Кбайт с 8-канальной ассоциативностью для каждого ядра;
- кеш-память данных уровня L1 на 32 Кбайт с 8-канальной ассоциативностью для каждого ядра;
- кеш-память уровня L2 на 8 Мбайт;
- кеш-память уровня L3 на 64 Мбайт;
- 20 линий PCIe 4.0;
- общая рассеиваемая мощность 105 Вт.

На момент выпуска процессор Ryzen 9 5950X, возможно, был самым производительным процессором семейства *x*86, доступным на рынке игр и фанатов производительности.

Прогнозирование ветвления в Ryzen 9 5950X

Архитектура Zen 3 включает в себя сложный блок прогнозирования ветвления, который кеширует информацию, описывающую выбранные ветви, и использует эти данные для повышения точности будущих прогнозов. Этот анализ не только охватывает отдельные ветви, но и оценивает корреляцию с недавними ветвями в соседнем коде для дальнейшего повышения точности прогнозирования. Повышенная точность прогнозирования помогает исключить снижение производительности из-за конвейерных пузырей и сводит к минимуму ненужную работу, связанную с упреждающим выполнением ветвей, которые в конечном счете не выбираются.

Блок прогнозирования ветвления использует форму машинного обучения, называемую **персептроном**. Персептроны — это упрощенные модели биологических нейронов, которые формируют основу для многих вариантов применения искусственных нейронных сетей. Краткое введение в искусственные нейронные сети было представлено в *разд. "Глубокое обучение" главы 6*.

В процессоре 5950X персептроны учатся предсказывать поведение ветвления отдельных инструкций на основе их недавнего поведения, а также поведения ветвления других инструкций. Таким образом, отслеживая характер недавних случаев ветвления (с точки зрения выбранных и невыбранных ветвей), можно разработать корреляции для рассматриваемой инструкции ветвления, которые помогают повысить точность прогнозирования.

Графический процессор Nvidia GeForce RTX 3090

Для процессора Aurora Ryzen Edition R10 предлагается опция в виде графического процессора Nvidia GeForce RTX 3090. В дополнение к общему высокому уровню графической производительности, которой можно ожидать от топового игрового графического процессора, это устройство обеспечивает мощную аппаратную поддержку трассировки лучей и содержит выделенные ядра для ускорения работы приложений машинного обучения.

В традиционных графических процессорах визуальные объекты описываются в виде наборов многоугольников. Для визуализации сцены сначала необходимо определить местоположение и пространственную ориентацию каждого многоугольника, после чего выполняется отрисовка видимых в сцене многоугольников в соответствующем месте изображения.

Трассировка лучей использует альтернативный, более сложный подход. Изображение с трассировкой лучей рисуется путем отслеживания пути света, излучаемого одним или несколькими источниками освещения в виртуальном мире. Когда лучи света встречаются с объектами, возникают такие эффекты, как отражение, преломление, рассеяние и образование теней.

Изображения с трассировкой лучей обычно выглядят гораздо более реалистичными, чем сцены с визуализацией на основе многоугольников, однако для трассировки лучей требуются гораздо большие вычислительные ресурсы.

Сегодня большинство популярных, визуально насыщенных, высокодинамичных игр хотя бы в малой степени используют трассировку лучей. Для разработчиков игр ее использование не является бескомпромиссным решением. Части сцен можно отрисовывать в традиционном режиме на основе многоугольников, а трассировку лучей использовать для визуализации объектов и поверхностей, когда преимущества этого метода позволяют извлечь наибольшую выгоду для отображаемой сцены. Например, сцена может содержать фоновые изображения, отображаемые в виде многоугольников, в то время как в расположенном рядом стеклянном окне с помощью метода трассировки лучей отображаются отражения объектов от поверхности стекла вместе с видом за стеклом.

На момент своего выпуска RTX 3090 был самым производительным графическим процессором, способным запускать модели глубокого обучения с помощью **TensorFlow** — популярной программной платформы с открытым исходным кодом для приложений машинного обучения, разработанной исследовательской организа-

цией Google по машинному интеллекту. TensorFlow широко применяется в исследованиях с использованием глубоких нейронных сетей.

RTX 3090 задействует свои возможности машинного обучения для увеличения видимого разрешения создаваемых изображений без вычислительных затрат на визуализацию объектов с повышенным разрешением. Это осуществляется путем интеллектуального применения к изображению эффектов сглаживания и повышения резкости. Эта технология изучает характеристики десятков тысяч изображений во время их визуализации и использует эту информацию для улучшения качества сцен, создаваемых впоследствии. Например, с помощью данной технологии сцена, отрисованная с разрешением 1080p (1920 × 1080 пикселов), может выглядеть так, как если бы она имела разрешение 1440p (1920 × 1440 пикселов).

Помимо поддержки технологий трассировки лучей и машинного обучения RTX 3090 обладает следующими особенностями.

- **Ядра 10496 NVIDIA CUDA®** составляют платформу параллельных вычислений, подходящую для решения общих вычислительных задач, таких как линейная алгебра.
- **328 тензорных ядер** выполняют тензорные и матричные операции в алгоритмах глубокого обучения.
- **Интерфейс PCIe 4.0 x16** обеспечивает взаимодействие с центральным процессором.
- **24 Гбайт памяти GDDR6X**, которая является усовершенствованием предыдущего поколения технологии GDDR6, обеспечивая повышенную скорость передачи данных (до 21 Гбит/с на контакт по сравнению с максимумом 16 Гбит/с на контакт в GDDR6).
- **Масштабируемый интерфейс обмена данными Nvidia Scalable Link Interface (SLI)** связывает от двух до четырех идентичных графических процессоров в системе для распределения рабочей нагрузки по обработке. Для соединения взаимодействующих графических процессоров необходимо использовать специальный мостовой разъем.
- **Три видеовыхода DisplayPort 1.4a.** Интерфейсы DisplayPort поддерживают разрешение 8K (7680 × 4320 пикселов) при частоте 60 Гц.
- **Порт HDMI 2.1.** Выход HDMI поддерживает разрешение 4K (3840 × 2160 пикселов) при частоте 60 Гц.

В следующем разделе кратко описываются подсистемы ПК Alienware Aurora Ryzen Edition R10.

Подсистемы ПК Aurora

В табл. 13.2 приведен краткий обзор основных подсистем ПК Alienware Aurora Ryzen Edition R10.

Таблица 13.2. Подсистемы ПК Alienware Aurora Ryzen Edition R10

Подсистема	Описание
Материнская плата	Материнская плата поддерживает интерфейс PCIe 4.0, удваивающий пропускную способность для обмена данными между процессором и видеокартой относительно PCIe 3.0. Имеется четыре гнезда для модулей памяти DDR4. Предусмотрены четыре гнезда PCIe, однако два из них занимает плата графического процессора Nvidia двойной ширины
Чипсет	Чипсет AMD B550A поддерживает разгон процессора и памяти, а также интерфейс PCIe 4.0
Охлаждение	Для охлаждения процессора используется система жидкостного охлаждения Alienware, которая критически необходима при разгоне
Память	Система включает в себя до 32 Гбайт двухканальной памяти DDR4 XMP, работающей на частоте 3200 МГц. Средства настройки конфигурации **Extreme Memory Profiles** (XMP) позволяют одновременно изменять несколько параметров, определяющих рабочие характеристики памяти, простым выбором среди различных профилей. Эта возможность обычно используется для выбора между стандартной конфигурацией синхронизации памяти и конфигурацией для разгона
Хранение данных	ПК Aurora Ryzen Edition R10 оснащен твердотельным накопителем NVMe M.2 емкостью 1 Тбайт. Стандартным интерфейсом для подсоединения твердотельных накопителей к PCIe 4.0 является **Non-Volatile Memory express** (NVMe). Стандарт M.2 определяет малый типоразмер для карт расширения, таких как твердотельные накопители
Передняя панель	Три порта USB 3.2 поколения 1 типа A, порт USB 3.2 поколения 1 типа C, а также разъемы аудиовхода и аудиовыхода
Задняя панель	Система включает в себя шесть портов USB 2.0, четыре порта USB 3.2 поколения 1 типа A, порт USB 3.2 поколения 1 типа C, порт Ethernet, а также разъемы цифровых и аналоговых аудиовходов и аудиовыходов

Игровой настольный ПК Alienware Aurora Ryzen Edition R10 объединяет в себе самые передовые технологии, доступные на момент его появления, по чистой скорости процессора, памяти, графического процессора и хранилища, а также технологию машинного обучения для ускорения выполнения команд.

В следующем разделе мы перейдем от уровня персонального компьютера, рассмотренного в этом разделе, к проблемам внедрения и проектным решениям, характерным для крупномасштабных вычислительных сред, состоящих из тысяч интегрированных и взаимодействующих друг с другом компьютерных систем.

Вычислительная архитектура масштаба центра обработки данных

Поставщики услуг доступа к крупномасштабным вычислительным и сетевым ресурсам для населения и разветвленных организаций, таких как правительственные учреждения, исследовательские университеты и крупные корпорации, часто объединяют вычислительные ресурсы в больших зданиях, каждое из которых может содержать тысячи компьютеров.

Для того чтобы наиболее эффективно использовать эти ресурсы, недостаточно рассматривать такую совокупность компьютеров в **вычислительном комплексе масштаба центра обработки данных** (warehouse-scale computer, WSC), как просто большое количество отдельных компьютеров. Вместо этого, принимая во внимание огромный объем вычислительных и сетевых ресурсов, а также ресурсов для хранения данных, предоставляемых WSC, более уместно рассматривать весь центр обработки данных (ЦОД) как единую вычислительную систему с массовым параллелизмом.

Ранние электронные вычислительные машины представляли собой огромные системы, занимавшие большие помещения. С тех пор компьютерные архитектуры эволюционировали до современных процессорных чипов размером с ноготь, обладающих значительно большей вычислительной мощностью, чем те ранние системы. Можно представить, что сегодняшние WSC — это прелюдия к компьютерным системам, которые через несколько десятилетий будут иметь размер с коробку из-под пиццы или смартфон, или даже ноготь, и обладать такой же вычислительной мощностью, как современные WSC, если не намного большей.

Со времени взлета Интернета в середине 1990-х годов происходит переход от программ, установленных на персональных компьютерах, к централизованным серверным системам, которые выполняют алгоритмические вычисления, сохраняют и извлекают огромные объемы данных и обеспечивают прямую связь между пользователями Интернета.

Эти серверные приложения используют тонкий уровень приложений на стороне клиента, часто предоставляемый веб-браузером. Все операции по поиску данных, вычислительной обработке и подготовке информации к отображению выполняются на сервере. Клиентское приложение просто получает инструкции и данные, касающиеся текста, графики и элементов управления пользовательским интерфейсом, для представления пользователю. Затем браузерное приложение ожидает ввода данных пользователем и отправляет запросы на действия обратно на сервер.

Онлайн-сервисы, предоставляемые такими интернет-компаниями, как Google, Amazon и Microsoft, полагаются на мощь и универсальность вычислительных архитектур очень крупных центров обработки данных для предоставления услуг миллионам пользователей. В одном из таких WSC можно запустить небольшое количество весьма крупных приложений, одновременно предоставляющих услуги тысячам пользователей.

Поставщики услуг стремятся обеспечить исключительную надежность, часто обещая время безотказной работы на уровне 99,99%, что соответствует примерно одному часу простоя в год.

В следующих подразделах представлены аппаратные и программные компоненты типичного WSC и обсуждается, как эти компоненты работают вместе, чтобы предоставлять быстрые, эффективные и высоконадежные интернет-сервисы большому количеству пользователей. Данный раздел завершается пошаговым описанием действий по созданию и развертыванию простого веб-приложения в коммерческой облачной среде.

Аппаратные средства WSC

Создание, эксплуатация и техническое обслуживание WSC — дорогостоящие мероприятия. Обеспечивая необходимое качество обслуживания (отражаемое такими показателями, как скорость отклика, пропускная способность по передаче данных и надежность), операторы WSC стремятся свести к минимуму общую стоимость владения и эксплуатации этих систем.

Для достижения максимально высокой надежности разработчики WSC могут использовать один из двух следующих подходов при выборе базовых аппаратных средств.

- **Вложение средств в оборудование, обладающее исключительной надежностью.** Этот подход основан на использовании дорогостоящих компонентов с низкой частотой отказов. Однако, даже если каждая отдельная компьютерная система обладает превосходной надежностью, со временем, когда несколько тысяч копий системы будут работать одновременно, случайные сбои будут происходить со статистически предсказуемой частотой. Этот очень дорогой подход, который в конечном счете не решает проблему, потому что сбои продолжают возникать.
- **Использование недорогого оборудования со средней надежностью и проектирование системы таким образом, чтобы она допускала отказы отдельных компонентов с максимально высокой ожидаемой частотой.** Такой подход позволяет значительно снизить затраты на оборудование по сравнению с высоконадежными компонентами, хотя для него требуется сложная программная инфраструктура, способная обнаруживать аппаратные сбои и быстро компенсировать их с помощью резервных систем таким образом, чтобы поддерживать заявленное качество обслуживания.

Большинство поставщиков стандартных интернет-услуг, таких как поисковые системы и службы электронной почты, используют недорогое универсальное вычислительное оборудование и при возникновении сбоев выполняют **переключение на резерв** путем перевода рабочих нагрузок на включенные резервные системы.

Для того чтобы конкретизировать это обсуждение, мы рассмотрим рабочие нагрузки, которые WSC должен поддерживать для функционирования в качестве поиско-

вой системы в Интернете. Рабочие нагрузки WSC, поддерживающие поиск в Интернете, должны обладать перечисленными ниже атрибутами.

- **Быстрое реагирование на поисковые запросы.** Обработка запроса на поиск в Интернете на стороне сервера должна занимать лишь малую долю секунды. Если пользователи будут регулярно сталкиваться с ощутимой задержкой, свои будущие запросы они, скорее всего, доверят конкурирующей поисковой системе.
- **Информацию о состоянии, относящуюся к каждому поисковому запросу, не требуется сохранять на сервере, даже при последовательных случаях взаимодействия с одним и тем же пользователем.** Другими словами, обработка каждого поискового запроса представляет собой завершенное взаимодействие. По завершении поиска сервер полностью забывает о нем. При следующем поисковом запросе в тот же сервис от того же пользователя никакая сохраненная информация из предыдущего запроса не используется.

Учитывая эти атрибуты, каждый запрос на обслуживание можно рассматривать как изолированное событие, не зависящее от всех других запросов, прошлых, настоящих и будущих. Независимость каждого запроса означает, что он может обрабатываться как поток выполнения параллельно с другими поисковыми запросами, поступающими от других пользователей или даже от того же самого пользователя. Эта модель рабочей нагрузки является идеальным кандидатом для ускорения за счет аппаратного параллелизма.

Обработка поисковых запросов в Интернете — это задача, которая требует не интенсивных вычислений, а интенсивной работы с данными. Простой пример: при выполнении поиска по запросу, состоящему из одного слова, веб-сервис получает запрос от пользователя, затем извлекает искомый термин и сверяется со своим индексом, чтобы определить наиболее подходящие для данного термина страницы.

Интернет содержит как минимум сотни миллиардов страниц, и пользователи ожидают, что смогут найти большинство из них с помощью поиска. Однако это чрезмерное упрощение, поскольку значительная доля страниц, доступных через Интернет, не индексируется поисковыми системами. Но даже если ограничить поиск доступными страницами, один сервер, пусть и с большим количеством процессорных ядер и максимальным установленным объемом оперативной и дисковой памяти, просто не сможет в разумные сроки справиться с ответами на запросы в Интернете для большой базы пользователей. Слишком много данных и слишком много пользовательских запросов. Вместо этого функцию поиска следует разделить между многими (сотнями, возможно, тысячами) отдельными серверами, каждый из которых содержит подмножество всего индекса веб-страниц, известных поисковой системе.

Каждый сервер индексирования получает поток запросов на поиск, которые прошли фильтрацию и относятся к той части индекса, которой он управляет. Сервер индексирования генерирует набор результатов на основе совпадений с искомым термином и возвращает этот набор для обработки на более высоком уровне. При более сложных поисковых запросах разные серверы индексирования могут выпол-

нять отдельные запросы по нескольким искомым терминам. Результаты этих поисков затем нужно отфильтровать и объединить во время обработки на более высоком уровне.

По мере того как серверы индексирования генерируют результаты на основе поисковых запросов, полученные подмножества передаются в систему, которая перерабатывает информацию в форму, подходящую для передачи пользователю. При стандартном поиске пользователи ожидают получить список страниц, ранжированных по степени их соответствия запросу. Для каждой возвращаемой страницы поисковая система обычно предоставляет URL-адрес целевой страницы вместе с фрагментом текста, содержащим искомый термин, чтобы предоставить пользователю некоторый контекст.

Время, необходимое для формирования этих результатов, в значительной степени зависит от скорости поиска в базе данных, связанной с индексом страницы, и извлечения содержимого страницы из хранилища, чем от вычислительной мощности серверов, участвующих в выполнении этой задачи. По этой причине многие WSC, предоставляющие сервисы веб-поиска и аналогичные услуги, используют серверы, содержащие недорогие материнские платы, процессоры, компоненты памяти и накопители.

Стоечные серверы

Серверы WSC обычно собираются в стойках, причем каждый сервер использует один отсек типоразмера 1U. Серверный отсек типоразмера 1U имеет отверстие на передней панели стойки шириной 433 мм (19 дюймов) и высотой 44,5 мм (1,75 дюйма). Одна стойка может содержать до 40 серверов, занимающих 178 см (70 дюймов) пространства по вертикали.

Каждый сервер представляет собой полнофункциональную компьютерную систему, содержащую процессор средней мощности, оперативную память, локальный накопитель и интерфейс Ethernet со скоростью от 1 Гбит/с. Поскольку возможности и пропускная способность процессоров потребительского класса, динамической памяти и накопителей продолжают расти, мы не будем пытаться определить параметры производительности конкретной конфигурации системы.

Несмотря на то что каждый сервер содержит процессор со встроенной графической подсистемой и несколько USB-портов, большинство серверов не имеют дисплея, клавиатуры или мыши, подключенных напрямую, за исключением, разве что, периода их первоначальной настройки. Серверы, устанавливаемые в стойку, обычно работают в режиме **удаленного управления**, в котором все взаимодействие с системой осуществляется через ее сетевое подключение.

На рис. 13.2 показана стойка, содержащая 16 серверов.

Каждый сервер подключается к гигабитному порту сетевого коммутатора стойки с помощью кабеля Ethernet. Стойка в этом примере подключается к сетевой среде WSC более высокого уровня с помощью четырех кабелей Ethernet со скоростью 1 Гбит/с. Серверы в стойке обмениваются данными друг с другом через коммута-

тор стойки на полной скорости передачи данных Ethernet 1 Гбит/с. Поскольку стойка имеет только четыре восходящих соединения со скоростью 1 Гбит/с, все 16 серверов, очевидно, не могут взаимодействовать на полной скорости с внешними по отношению к стойке системами. В этом примере подключение стойки является **перегруженным** с коэффициентом 4. Это означает, что пропускная способность внешней сети составляет одну четверть от пиковой скорости обмена данными для серверов в стойке.

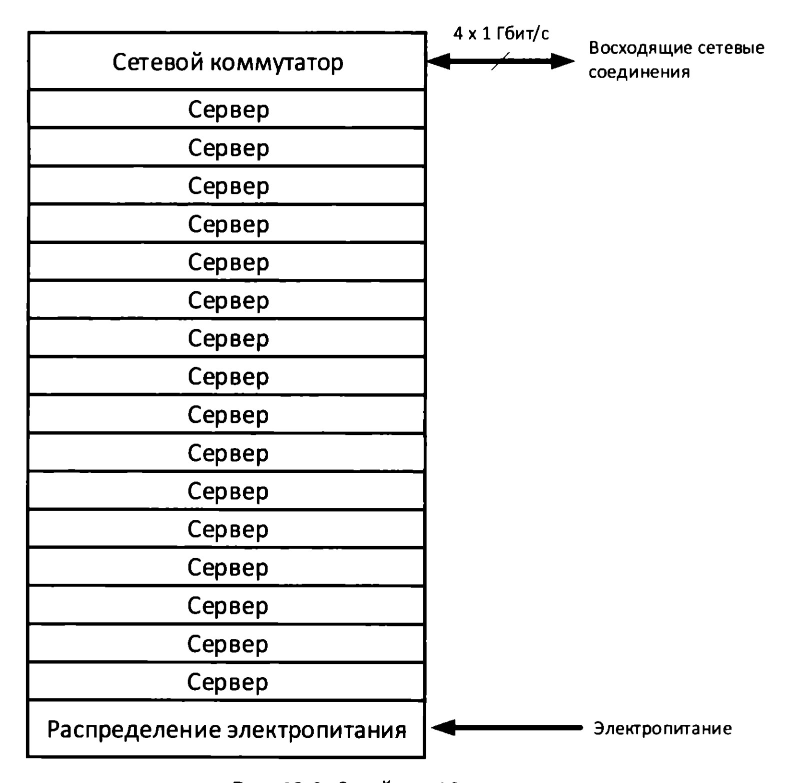


Рис. 13.2. *Стойка с 16 серверами*

Стойки организованы в кластеры, которые совместно используют кластерный коммутатор второго уровня. На рис. 13.3 показана конфигурация, в которой четыре стойки подключены к каждому коммутатору кластерного уровня, который, в свою очередь, подключен к сети уровня WSC.

В конфигурации WSC, показанной на рис. 13.3, запрос пользователя поступает через Интернет и первоначально обрабатывается маршрутизатором, который направляет запрос на доступный веб-сервер. Сервер, получающий запрос, отвечает за контроль процесса поиска и отправку ответа обратно пользователю.

Несколько веб-серверов постоянно подключены к сети, чтобы обеспечить распределение нагрузки и резервирование в случае сбоя. На рис. 13.3 показаны три веб-сервера, но в загруженном WSC могут одновременно работать гораздо больше серверов. Веб-сервер анализирует поисковый запрос и перенаправляет его на соответствующие серверы индексирования в кластерах стоек WSC. На основе искомых

терминов веб-сервер направляет запросы на поиск по индексу одному или нескольким серверам индексирования для обработки.

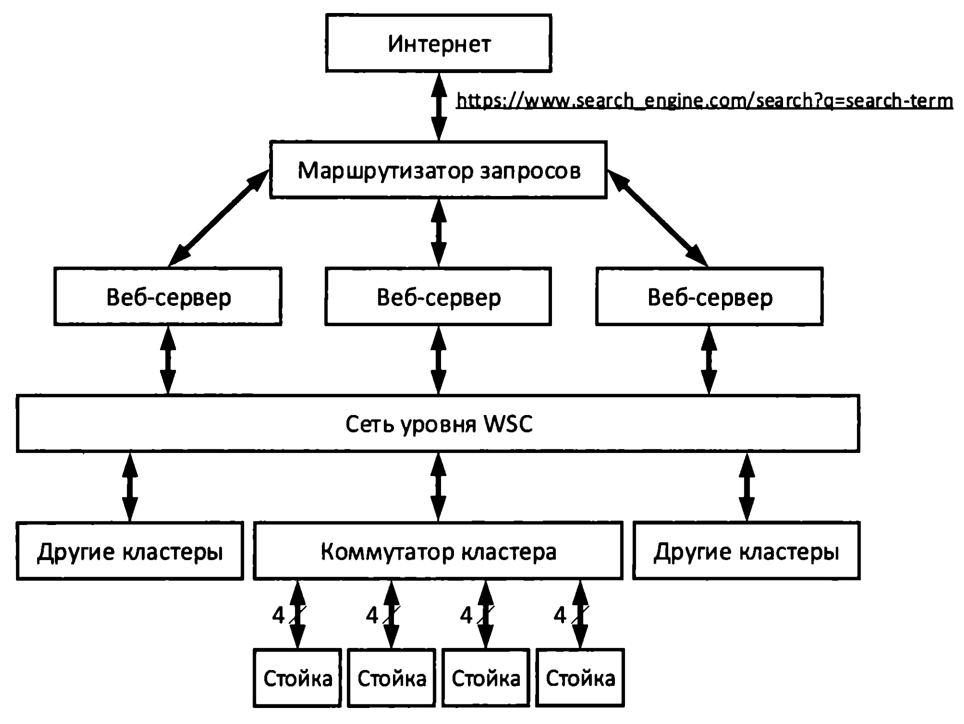


Рис. 13.3. *Внутренняя сеть WSC*

Для эффективной и надежной работы WSC должен поддерживать несколько копий каждого подмножества базы данных индексирования, распределенных между несколькими кластерами, чтобы обеспечить распределение нагрузки и резервирование в случае сбоев на уровне сервера, стойки или кластера.

Запросы на поиск по индексу обрабатываются серверами индексирования, а соответствующий текст целевой страницы собирается с серверов документов. Полный набор результатов поиска собирается и передается обратно на соответствующий веб-сервер. Затем веб-сервер подготавливает полный ответ и передает его пользователю.

Конфигурация реального WSC будет сложнее тех схем, что показаны на рис. 13.2 и 13.3. Тем не менее эти упрощенные представления помогают нам оценить некоторые из важных возможностей WSC, поддерживающих рабочую нагрузку поисковой системы в Интернете.

Помимо реагирования на поисковые запросы пользователей поисковая система должна регулярно обновлять свою базу данных, чтобы сохранить актуальность относительно текущего состояния веб-страниц в Интернете. Поисковые системы обновляют свои знания о веб-страницах с помощью приложений, называемых **поисковыми роботами**. В качестве отправной точки поисковый робот выбирает адрес веб-страницы, затем считывает целевую страницу и анализирует ее содержимое.

Далее он сохраняет текст страницы в базе данных документов поисковой системы и извлекает все содержащиеся в ней ссылки. Для каждой найденной ссылки поисковый робот повторяет процесс чтения страницы, синтаксического анализа и перехода по ссылке. Таким образом, поисковая система создает и обновляет свою индексированную базу данных интернет-контента.

В этом подразделе была кратко описана концепция конфигурации вычислительного комплекса уровня ЦОД (WSC), который основан на стойках, заполненных стандартными вычислительными компонентами. В следующем разделе рассматриваются меры, которые WSC должен предпринять для обнаружения отказов компонентов и компенсации этих отказов без ущерба для общего качества обслуживания.

Управление отказами аппаратных средств

Как было показано выше, WSC содержат тысячи компьютерных систем. Мы вполне можем ожидать, что аппаратные сбои будут происходить регулярно, даже если для системы были выбраны наиболее дорогостоящие компоненты, обеспечивающие высокий, но не идеальный уровень надежности.

На схеме, показанной на рис. 13.3, подразумевается, что каждый сервер, отправляющий запрос в систему на более низком уровне, должен отслеживать скорость реагирования и корректность откликов системы, назначенной для обработки запроса. Этот процесс является неотъемлемой частью многоуровневых операций отправки, обработки и возврата результатов. Если ответ недопустимо задерживается или не проходит проверку на достоверность, система нижнего уровня должна быть зарегистрирована как не реагирующая на запросы или неправильно работающая.

При обнаружении такой ошибки запрашивающая система сразу же пересылает запрос на резервный сервер для обработки. В некоторых случаях задержки откликов могут быть вызваны временны́ми событиями, такими как кратковременная перегрузка ресурсов обработки. Если сервер нижнего уровня восстанавливается и продолжает работать должным образом, никакие ответные действия не требуются.

Если же сервер постоянно не отвечает на запросы или выдает ошибки, необходимо отправить запрос на техническое обслуживание для устранения неполадок и ремонта неисправной системы. При выявлении недоступной системы руководство WSC (будь то человек или автоматическая система) может отдать команду на выбор другой системы из пула резервных систем для репликации отказавшего сервера и дать указание сменной системе начать обслуживание запросов от пользователей.

Потребление электроэнергии

Одним из основных факторов, определяющих стоимость WSC, является потребление электроэнергии. Основными потребителями электроэнергии в WSC являются серверы и сетевые устройства, которые выполняют обработку данных для конечных пользователей, а также система кондиционирования воздуха, обеспечивающая охлаждение этих систем.

Для того чтобы свести счета за электроэнергию WSC к минимуму, крайне важно включать компьютеры и другие потребляющие много энергии устройства только тогда, когда они могут сделать что-то полезное. Нагрузка на поисковую систему с течением времени сильно меняется и может резко увеличиваться в ответ на события в новостях и социальных сетях. WSC должен поддерживать достаточное количество рабочих серверов, чтобы обеспечить максимальный уровень трафика, на который он рассчитан. Когда общая рабочая нагрузка ниже максимальной, все серверы, для которых нет работы, должны быть выключены.

Даже слегка загруженный сервер потребляет значительное количество электроэнергии. Для достижения наилучшей эффективности система управления WSC должна полностью отключать серверы и другие устройства, когда они не нужны. При увеличении трафика серверы и связанные с ними сетевые устройства могут быть быстро включены и подсоединены к сети для поддержания требуемого качества обслуживания.

WSC как многоуровневый информационный кеш

Мы рассмотрели многоуровневую архитектуру кеша, используемую в современных процессорах, в *главе 8*. Для достижения оптимальных рабочих характеристик веб-сервис, такой как поисковая система, должен использовать стратегию кеширования, которая, по сути, добавляет дополнительные уровни к тем, что уже существуют в процессоре.

Для того чтобы свести время отклика к минимуму, сервер индексирования должен хранить значительное подмножество своих индексных данных в базе данных, находящейся в памяти. Выбирая контент для хранения в памяти на основе исторических моделей использования, а также последних тенденций поиска, можно добиться удовлетворения значительной доли входящих запросов без доступа к дисковому хранилищу.

Для того чтобы наилучшим образом использовать базу данных в памяти, явно выгодно иметь большой объем DRAM на каждом сервере. Выбор оптимального объема DRAM для установки на каждый сервер индексирования зависит от таких параметров, как относительная стоимость дополнительной DRAM на сервер по сравнению со стоимостью дополнительных серверов, содержащих меньше памяти, а также от сравнения производительности большего количества серверов с меньшим объемом памяти и меньшего количества серверов с большим объемом памяти.

Мы не будем здесь углубляться в этот анализ, однако следует отметить, что такие оценки являются ключевым элементом оптимизации архитектуры WSC.

Если мы рассматриваем DRAM как первый уровень кеширования на уровне WSC, то следующий уровень — это локальный накопитель на каждом сервере. В случае отсутствия нужной информации в базе данных в памяти следующим местом для поиска является накопитель сервера. Если результат не найден на локальном накопителе, то следующий уровень поиска — это другие серверы, расположенные в той

же стойке. Обмен данными между серверами в одной стойке может осуществляться на полной скорости сети (1 Гбит/с в нашем примере конфигурации).

Следующий уровень поиска охватывает стойки в пределах одного кластера. Пропускная способность передачи данных между стойками определяется связывающими их линиями и характеристиками кластерного коммутатора, которые ограничивают скорость обмена данными по этим соединениям. Конечный уровень поиска в WSC охватывает кластеры, что, вероятно, накладывает дополнительные ограничения на пропускную способность.

Значительная часть задач по созданию эффективной инфраструктуры поисковой системы заключается в разработке высокопроизводительной архитектуры программного обеспечения. Эта архитектура должна удовлетворять высокую долю поисковых запросов с помощью самых быстрых и локализованных поисковых запросов, доступных серверам индексирования и серверам документов поисковой системы. Это означает, что основная доля поисковых запросов должна выполняться с помощью поиска в памяти на серверах индексирования.

Развертывание облачного приложения

В этом разделе мы рассмотрим необходимые действия по разработке и развертыванию простого веб-приложения на облачной платформе Microsoft Azure. Этот пример показывает, как разработчики могут в полной мере использовать преимущества операторов крупных и сложных WSC, применяя легкодоступные инструменты разработки. В этом примере используются бесплатные программные средства и облачный сервис.

1. Перейдите по адресу **https://azure.microsoft.com/en-us/free/** и создайте бесплатную учетную запись Azure. Azure — это сервис облачных вычислений, предоставляемый корпорацией Microsoft.
2. Перейдите по адресу **https://nodejs.org/en/** и установите Node.js — среду выполнения для веб-серверных приложений, написанных на языке JavaScript.
3. Перейдите по адресу **https://code.visualstudio.com/** и установите Visual Studio Code (сокращенно VS Code) — многоязычный редактор исходного кода.
4. Перейдите по адресу **https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azureappservice** и установите Azure App Service для VS Code. Azure App Service — это расширение для VS Code, которое помогает создавать и развертывать веб-приложения в Azure.

Приведенные ниже инструкции предназначены для демонстрации в операционной системе Windows, но аналогичные (или идентичные во многих случаях) команды будут работать и на хостах с ОС Linux.

После установки этих программ откройте окно командной строки Windows и введите следующие команды, чтобы создать простое веб-приложение и запустить его на своем компьютере:

```
C:\Projects>npx express-generator webapp --view pug
C:\Projects>cd webapp
C:\Projects\webapp>npm install
C:\Projects\webapp>npm start
```

Откройте веб-браузер и перейдите по адресу **http://localhost:3000**. Вы увидите веб-страницу, которая выглядит так, как показано на рис. 13.4.

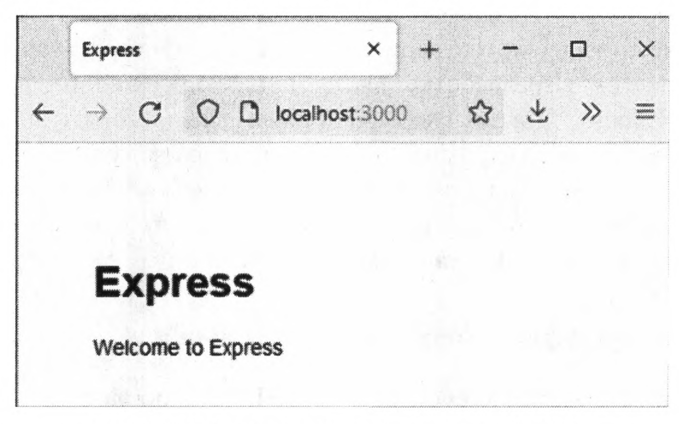


Рис. 13.4. *Экран простого приложения Node.js*

Это приложение не делает ничего объективно полезного, однако код, созданный на предыдущих этапах, составляет прочную основу для создания сложного, масштабируемого веб-приложения.

На следующих этапах мы развернем это приложение в облачной среде Azure, используя бесплатную учетную запись Azure.

1. Запустите редактор VS Code в каталоге webapp:

```
C:\Projects\webapp>code
```

2. Выберите логотип Azure, показанный в левом нижнем углу на рис. 13.5.
3. Нажмите **Sign in to Azure...** (Вход в Azure) и завершите процесс создания новой бесплатной учетной записи и входа в систему.
4. Щелкните на значке облака справа от **APP SERVICE**, как показано на рис. 13.6. Если значок не виден, наведите курсор на эту область, и он появится. При появлении приглашения перейдите в папку **webapp**.
5. Щелкните + | **Create new Web App...** | **Advanced** (+ | Создать новое веб-приложение... | Дополнительно). Появляется приглашение ввести уникальное имя для своего приложения. Например, имя **webapp** уже используется, но **webapp228** — еще нет. Выберите другое имя.
6. Щелкните + | **Create a new resource group** (+ | Создать группу ресурсов) и дайте группе имя, например **webapp228-rg**.

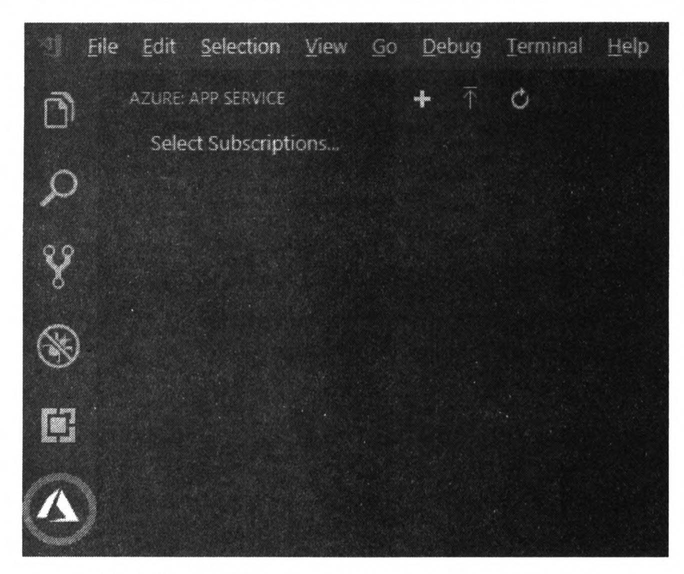


Рис. 13.5. *Добавление параметра приложения*

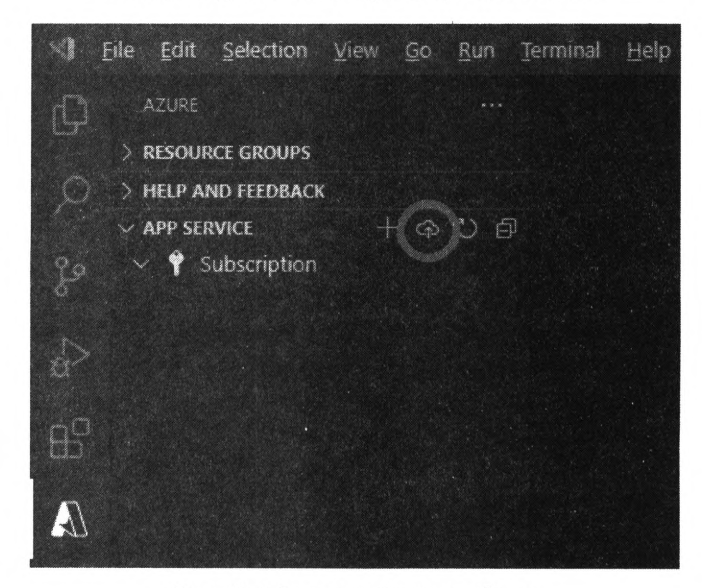


Рис. 13.6. *Развертывание в облаке*

7. Появляется приглашение выбрать стек времени выполнения. В этом случае выберите **Node 16 LTS**.
8. Выберите операционную систему **Windows**.
9. Выберите географическое местоположение для развертывания, например **West US 2**.
10. Щелкните **+ | Create new App Service plan** (+ | Создать новый план служб приложений). Дайте плану имя, например **webapp228-plan**. Выберите уровень цен **Free (F1)** (Бесплатно).

11. При появлении приглашения **+ Create new Application Insights** (Создать Application Insights) выберите **Skip for now** (Пока пропустить).

12. В Azure начинается подготовка приложений, которая займет некоторое время. При появлении приглашения **Always deploy the workspace "webapp" to "webapp228"?** (Всегда развертывать рабочее пространство webapp в webapp228?) выберите **Yes** (Да).

13. В VS Code раскройте узел **APP SERVICE**, затем раскройте **webapp228**. Щелкните правой кнопкой мыши на команде **Application Settings** (Настройки приложения). Выберите команду **Add New Setting...** (Добавить новое значение...), как показано на рис. 13.7.

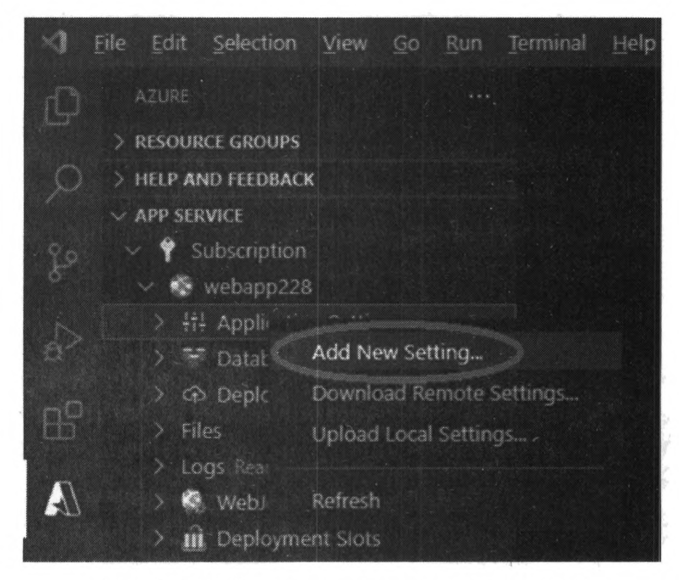


Рис. 13.7. *Добавление параметра приложения*

14. Введите `SCM_DO_BUILD_DURING_DEPLOYMENT` в качестве ключа параметра и `true` в качестве его значения. Этот шаг приведет к автоматическому созданию файла `web.config`, который необходим для развертывания.

15. Снова выберите значок облака рядом с **APP SERVICE** и разверните приложение.

16. После завершения развертывания щелкните на **Browse Website** (Обзор веб-сайта), чтобы открыть веб-сайт в своем браузере. В этом примере веб-сайт имеет URL-адрес **https://webapp228.azurewebsites.net/**. Это публичный веб-сайт, доступный любому пользователю Интернета.

Описанные выше действия продемонстрировали процедуру создания и развертывания простого веб-приложения в полнофункциональной облачной среде. Благодаря развертыванию в среде Azure наше приложение в полной мере использует возможности облачной платформы Azure, которые обеспечивают высокую производительность, масштабируемость и безопасность.

В следующем разделе рассматриваются высокопроизводительные архитектуры, используемые в специализированных процессорах для нейронных сетей.

Архитектура процессоров для нейронных сетей и машинного обучения

С архитектурой нейронных сетей мы кратко ознакомились в *главе 6*. В этом разделе рассматривается внутреннее устройство высокопроизводительного специализированного процессора для нейронных сетей.

Нейропроцессор Intel Nervana

В 2019 г. корпорация Intel объявила о выпуске пары новых процессоров, один из которых был оптимизирован для задачи обучения сложных нейронных сетей, а другой — для использования обученных сетей с целью выполнения логического вывода — процесса получения выходных данных нейронной сети на основе заданного набора входных данных.

Нейропроцессор для обучения (neural network processor for training, NNP-T) по сути представляет собой миниатюрный суперкомпьютер, адаптированный к вычислительным задачам, необходимым для процесса обучения нейронной сети. NNP-T1000 доступен в следующих двух конфигурациях.

- NNP-T1300 представляет собой карту с двумя гнездами PCIe, подходящую для установки в стандартный ПК. Карта связывается с хостом через интерфейс PCIe 4.0 x16. Несколько карт NNP-T1300, установленных в одной или в нескольких компьютерных системах, можно соединить друг с другом с помощью кабелей.
- NNP-T1400 — это мезонинная карта для использования в качестве модуля обработки в модуле-ускорителе **Open Compute Project (OCP) Accelerator Module (OAM)**. **Мезонинная карта** — это печатная плата, которая вставляется в разъем на другой вставной печатной плате, например на плате PCIe. OAM — это спецификация проектирования аппаратных архитектур, реализующих системы искусственного интеллекта, которые требуют высокой пропускной способности для обмена данными между модулями. Разработкой стандарта OAM руководили Facebook, Microsoft и Baidu. Путем объединения до 1024 модулей NNP-T1000 можно создать крупную архитектуру нейросетевой обработки с чрезвычайно высокоскоростными последовательными соединениями между модулями.

NNP-T1300 можно установить в стандартный ПК для использования отдельным разработчиком. При этом конфигурация из нескольких процессоров NNP-T1400 быстро становится весьма дорогостоящей и начинает напоминать суперкомпьютер с точки зрения производительности.

Основные области применения для мощных архитектур NNP, таких как Nervana, — **обработка информации на естественном языке** (natural language processing, NLP) и машинное зрение. Технология NLP решает такие задачи, как обработка последовательностей слов для извлечения заложенного в них смысла, а также составление сообщений на естественном языке для взаимодействия компьютера с людьми. Когда вы звоните на линию поддержки клиентов компании и компьютер просит вас поговорить с ним, вы взаимодействуете с системой NLP.

Машинное зрение — это ключевая технология, позволяющая создавать автономные транспортные средства. Системы машинного зрения для автомобилей обрабатывают потоки данных с видеокамер для обнаружения и классификации дорожных объектов, дорожных знаков и препятствий, таких как транспортные средства и пешеходы. Для того чтобы быть полезной во время управления транспортным средством, эта обработка должна выдавать результаты в реальном времени.

Построение нейронной сети для выполнения задач, которые обычно решает человек, таких как чтение текста и его интерпретация или вождение автомобиля в условиях интенсивного движения, требует обширного обучения. Обучение нейронной сети включает в себя последовательные этапы предоставления сети набора входных данных вместе с ответом, который должна выдать сеть при наличии таких входных данных. Эта информация, состоящая из пар наборов входных данных и наборов заведомо правильных выходных данных, называется **обучающим набором**. Каждый раз, когда сеть получает новый набор входных данных и выходные данные, которые должны быть получены на основе этих входных данных, она слегка корректирует свои внутренние соединения и весовые коэффициенты, чтобы улучшить свою способность генерировать правильные выходные данные. Для сложных нейронных сетей, на которые нацелено решение Nervana NNP, обучающий набор может состоять из миллионов пар наборов входных и выходных данных.

Обработка, требуемая алгоритмами обучения NNP, сводится в основном к матричным и векторным манипуляциям. Умножение больших матриц является одной из наиболее распространенных и трудоемких задач при обучении нейронных сетей. Эти матрицы могут содержать сотни или даже тысячи строк и столбцов. Основной операцией при умножении матриц является умножение с накоплением (MAC), с которой мы ознакомились в *главе 6*.

Сложные нейронные сети содержат огромное количество весовых параметров. Во время обучения процессор должен раз за разом обращаться к этим значениям, чтобы вычислять силу сигнала, связанную с каждым нейроном в модели, и вносить коррективы в веса при обучении. Для достижения максимальной производительности при заданных объеме памяти и пропускной способности внутренней связи желательно использовать для хранения каждого числового значения наименьший применимый тип данных. В большинстве приложений обработки чисел 32-битный формат IEEE с плавающей запятой и одинарной точностью представляет собой наименьший тип данных. Если возможно, лучше использовать еще меньший формат с плавающей запятой.

В архитектуре процессора Nervana для хранения сигналов сети используется специализированный формат с плавающей запятой. Этот формат, **bfloat16**, основан на 32-битном формате IEEE-754 с плавающей запятой и одинарной точностью за исключением того, что мантисса усечена с 24 до 8 бит. 32-битный и 64-битный форматы IEEE-754 с плавающей запятой были рассмотрены в *разд. "Арифметика с плавающей запятой" главы 9*.

Причины использования для нейросетевой обработки формата bfloat16 вместо 16-битного формата IEEE-754 с плавающей запятой и половинной точностью заключаются в следующем.

- 16-битный формат IEEE-754 содержит знаковый бит, 5 бит порядка и 11 бит мантиссы, один из которых является подразумеваемым. По сравнению с 32-битным форматом IEEE-754 с плавающей запятой и одинарной точностью этот формат с половинной точностью теряет три бита порядка, уменьшая диапазон представляемых числовых значений до одной восьмой диапазона 32-битного представления с плавающей запятой.
- Формат bfloat16 сохраняет все 8 бит порядка, принятые в формате одинарной точности IEEE-754, что позволяет ему охватывать весь числовой диапазон 32-битного формата IEEE-754, хотя и со значительно сниженной точностью.

Основываясь на результатах исследований и отзывах клиентов, Intel предполагает, что формат bfloat16 лучше всего подходит для приложений глубокого обучения, поскольку больший диапазон порядка важнее, чем более точная мантисса. В сущности, Intel предполагает, что эффект квантования, возникающий в результате уменьшения размера мантиссы, не оказывает существенного влияния на точность логического вывода в сетевых реализациях на основе формата bfloat16 по сравнению с реализациями на основе IEEE-754 с одинарной точностью.

Основной тип данных, используемый при обработке данных в искусственных нейронных сетях, — это **тензор**, который представлен в виде многомерного массива. Вектор — это одномерный тензор, а матрица — двумерный тензор. Также могут быть определены тензоры более высокой размерности. В архитектуре процессора Nervana тензор представляет собой многомерный массив значений bfloat16. Тензор является основным типом данных архитектуры Nervana — NNP-T работает с тензорами на уровне набора инструкций.

Наиболее ресурсоемкой операцией, выполняемой алгоритмами глубокого обучения, является умножение тензоров. Ускорение этих операций умножения — основная цель специализированного оборудования для обработки данных искусственными нейронными сетями, такого как архитектура Nervana. Ускорение операций с тензорами требует не только высокопроизводительной математической обработки; также важно эффективно передавать данные операндов в ядро для обработки и столь же эффективно перемещать выходные результаты по месту их назначения. Это требует тщательной балансировки возможностей по обработке чисел, скорости чтения/записи в память и скорости передачи данных.

Обработка в архитектуре NNP-T выполняется в **кластерах тензорных процессоров** (tensor processor clusters, TPC), каждый из которых содержит два модуля вы-

полнения операций **умножения с накоплением** (multiply accumulate, MAC) и 2,5 Мбайт высокоскоростной памяти. **Высокоскоростная память** (high bandwidth memory, HBM) отличается от DDR SDRAM тем, что в ней несколько кристаллов DRAM собраны в пакет, благодаря чему обеспечивается гораздо более широкая полоса передачи данных (1024 бита по сравнению с 64 битами для DDR). Каждый модуль обработки MAC содержит массив 32 × 32 параллельно работающих MAC-процессоров.

Процессор NNP-T содержит до 24 параллельно работающих кластеров тензорных процессоров (TPC) с высокоскоростными последовательными интерфейсами, соединяющими их в конфигурацию типа "фабрика". Устройства Nervana обеспечивают высокоскоростные последовательные соединения между дополнительными платами Nervana в одной и той же системе и с устройствами Nervana на других компьютерах.

Один процессор NNP-T может выполнять 119 **триллионов операций (или тераопераций) в секунду** (trillion operations per second, TOPS). В табл. 13.3 сравниваются параметры этих двух процессоров.

Таблица 13.3. *Особенности процессорных конфигураций NNP T-1000*

Параметр	**NNP-T1300**	**NNP-T1400**
Типоразмер устройства	Карта двойной ширины, PCIe 4.0 x16	OAM 1.0
Процессорные ядра (кластеры тензорных процессоров)	22 TPC	24 TPC
Тактовая частота процессора	950 МГц	1100 МГц
Статическая оперативная память	55 Мбайт SRAM с ECC	60 Мбайт SRAM с ECC
Высокоскоростная память	32 Гбайт **высокоскоростной памяти второго поколения (HBM2)** с ECC	32 Гбайт HBM2 с ECC
Пропускная способность памяти	2,4 Гбит/с (300 Мбайт/с)	2,4 Гбит/с (300 Мбайт/с)
Последовательное **межчиповое соединение** (inter-chip link, ICL)	16 × 112 Гбит/с (448 Гбайт/с)	16 × 112 Гбит/с (448 Гбайт/с)

Нейропроцессор Nervana для логического вывода (neural network processor for inference, NNP-I) выполняет фазу логического вывода в процессе обработки данных нейронной сети. Фаза логического вывода состоит из ввода входных данных в предварительно обученные нейронные сети, обработки этих входных данных и получения выходных данных из сети. В зависимости от области применения процесс логического вывода может включать в себя повторяющиеся оценки одной очень

большой сети на основе меняющихся во времени входных данных или применение множества различных моделей нейронной сети к одному и тому же набору входных данных при каждом обновлении этих входных данных.

Нейропроцессор NNP-I доступен в двух следующих типоразмерах:

- карта PCIe с двумя устройствами NNP I-1000, которая рассчитана на 170 тераопераций в секунду и рассеивает до 75 Вт;
- карта M.2 с одним устройством NNP I-1000, которая рассчитана на 50 тераопераций в секунду и рассеивает всего 12 Вт.

Архитектура Nervana — это продвинутая среда обработки, подобная суперкомпьютеру и оптимизированная для обучения нейронных сетей и построения логических выводов на основе реальных данных с использованием предварительно обученных сетей.

Резюме

В этой главе представлено несколько архитектур компьютерных систем, адаптированных к особым потребностям пользователей, и определены некоторые ключевые особенности, связанные с каждой из них. Мы рассмотрели следующие категории применения: смартфоны, игровые персональные компьютеры, вычислительные комплексы уровня ЦОД и нейронные сети. Эти примеры обеспечили связь между теоретическими обсуждениями компьютерных и системных архитектур и компонентов, представленных в предыдущих главах, и реальными реализациями современных высокопроизводительных вычислительных систем.

Прочитав эту главу, вы должны понимать процессы принятия решений, используемых при определении компьютерных архитектур для поддержки конкретных потребностей пользователей. Вы также получили представление о ключевых требованиях, определяющих архитектуру интеллектуальных мобильных устройств, высокопроизводительных персональных компьютеров, вычислительных комплексов уровня ЦОД и машинного обучения.

В следующей главе представлены категории угроз кибербезопасности, с которыми сталкиваются современные компьютерные системы, и рассмотрены вычислительные архитектуры, подходящие для приложений, требующих исключительных гарантий безопасности, таких как системы национальной безопасности и обработка финансовых транзакций.

Упражнения

1. Нарисуйте блок-схему вычислительной архитектуры для системы круглосуточных измерений и передачи данных о погоде с интервалом в 5 минут с помощью текстовых сообщений SMS. Система работает от аккумулятора и использует солнечные батареи для подзарядки аккумулятора в светлое время суток. Предположим, что метеорологические приборы потребляют минимальную среднюю

мощность, требуя лишь кратковременного выхода на полную мощность в каждом цикле измерений.

2. Для системы из *упражнения 1* определите подходящий доступный на рынке процессор и укажите причины, по которым этот процессор является хорошим выбором для данного приложения. Факторы, которые следует учитывать, включают стоимость, скорость обработки, устойчивость к суровым условиям, энергопотребление и встроенные компоненты, такие как оперативная память и интерфейсы для обмена данными.

14

Архитектуры для обеспечения кибербезопасности и конфиденциальности вычислений

В этой главе представлены вычислительные архитектуры, подходящие для решения задач, требующих исключительных гарантий безопасности. Такой высокий уровень защиты необходим в критически важных областях, включая системы национальной безопасности и обработку финансовых транзакций. Эти системы должны быть устойчивы к широкому спектру угроз кибербезопасности, в том числе к проникновению вредоносного кода, атакам через скрытые каналы и путем физического доступа к аппаратным средствам компьютеров. Среди тем, рассматриваемых в этой главе, — угрозы кибербезопасности, шифрование, цифровые подписи и архитектурные решения для защиты аппаратных средств и программного обеспечения.

После прочтения этой главы вы будете способны определять категории угроз кибербезопасности, с которыми может столкнуться система, и разбираться в способах обеспечения безопасности современных компьютеров. Вы узнаете, как можно избежать брешей в безопасности системных архитектур и как защищенная компьютерная архитектура может помочь обеспечить безопасность в программных приложениях.

В этой главе будут представлены следующие темы:

- угрозы кибербезопасности;
- особенности защищенного оборудования;
- конфиденциальные вычисления;
- меры безопасности на уровне архитектуры;
- обеспечение безопасности системного и прикладного ПО.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Угрозы кибербезопасности

Первым шагом в обеспечении кибербезопасности компьютерной системы является понимание угроз, с которыми она может столкнуться и от которых должна быть защищена. Мы можем разделить эти угрозы на несколько широких категорий и определить ключевые особенности каждой из них. Используя эту информацию, мы можем спроектировать архитектуру компьютерной системы с атрибутами, которые обеспечат надлежащий уровень защиты от этих угроз.

К сожалению, разработка полностью защищенной компьютерной системы — крайне непростой процесс. Это связано с тем, что в существующих операционных системах, библиотеках программного обеспечения, пользовательских приложениях и веб-приложениях регулярно выявляются новые уязвимости. Нередко обнаруживаются недостатки в широко используемых криптографических компонентах, таких как алгоритмы шифрования и протоколы аутентификации. По мере разработки новых программных продуктов в них часто возникают совершенно новые уязвимости, которые рано или поздно будут обнаружены. Самое главное, опыт показывает, что наиболее слабые места в системе безопасности, связанные с используемыми нами компьютерными системами, создаются людьми, которые ими управляют.

Учитывая эти реалии, мы можем лишь стремиться снизить риск разрушительной кибератаки до достаточно низкого уровня, а не устранить его полностью. В процессе защиты системы от атак мы должны удерживать стоимость защитных мер на приемлемом уровне, при этом система не должна быть слишком обременительной для пользователей в эксплуатации. Средства защиты также не должны приводить к снижению надежности системы с точки зрения выполнения ее предполагаемой функции.

Категории угроз кибербезопасности

В качестве первого шага к пониманию ситуации в области угроз кибербезопасности мы рассмотрим категории людей и организаций, которые планируют и осуществляют кибератаки. На основании исторических моделей кибератак против отдельных лиц, государственных учреждений, предприятий и других групп были определены следующие основные категории угроз.

- **Национальные правительства.** Было достоверно определено, что многие страны, в том числе те, которые можно рассматривать как "враждебные", а также те, которые принято считать "дружественными", проводили агрессивные кибератаки против других государств, коммерческих предприятий и

иных организаций в разных странах и даже против конкретных лиц. Военные киберподразделения планируют и осуществляют кибератаки против враждебных государств для достижения тактических и стратегических целей.

- **Террористы.** Террористические организации выражали намерения провести потенциально разрушительные кибератаки против правительств и промышленных систем с различными возможными последствиями — от широкомасштабных отключений электроэнергии до разрушения крупномасштабных объектов инфраструктуры, таких как плотины, нефтеперерабатывающие заводы и трубопроводы. Потенциал таких атак был продемонстрирован неоднократно, однако реальные случаи разрушительных нападений до сих пор были редки.
- **Промышленные шпионы.** Во многих отраслях, особенно связанных с высокими технологиями, секретная информация о программных компонентах и аппаратных средствах может стать главной мишенью для хакеров. Если злоумышленники смогут получить доступ к компьютерной сети, где находится желаемая информация, они могут скопировать ее и использовать в собственных целях или просто продать тому, кто предложит наиболее высокую цену.
- **Преступные группировки.** Группы криминальных хакеров проводят атаки, такие как проникновение в компьютерные системы, содержащие личную информацию, например данные кредитных карт, с целью ее продажи другим преступникам или непосредственного использования для совершения незаконных покупок. Еще одна категория киберпреступлений — вымогательство с помощью вредоносных программ. При атаке с помощью программы-вымогателя хакеры загружают на компьютер жертвы программное обеспечение, которое шифрует файлы ценных данных. Злоумышленник требует от жертвы оплаты в обмен на предоставление ключа, который предположительно позволит расшифровать данные пользователя. На нижней ступени технологической лестницы продолжает процветать мошенничество с массовыми рассылками по электронной почте. Таким способом злоумышленники пытаются вовлечь получателей в разговор с преступниками, который может привести к дорогостоящему разочарованию для неосторожных.
- **Хакеры-активисты.** Некоторые кибер-злоумышленники заявляют, что в своих атаках против предположительно аморальных врагов, таких как неугодные корпорации или правительства, руководствуются справедливыми мотивами. Таких людей называют **хактивистами**, комбинируя слова *"хакер"* и *"активист"*. Хактивисты предпринимают такие действия, как массовая бомбардировка целевого веб-сайта огромным количеством запросов, в результате чего сайт становится недоступным для обычных пользователей. Хактивисты также могут попытаться украсть принадлежащие цели конфиденциальные данные, а затем опубликовать их, чтобы поставить в неловкое положение или скомпрометировать организацию, которая является источником этой информации.

- **Белые хакеры.** Следуя определенной традиции, хакеры могут исследовать цель, а затем спланировать и осуществить против нее кибератаку ради чистого интеллектуального удовольствия. В других случаях исследователи кибербезопасности могут изучать цель, возможно, веб-сайт или цифровое устройство, такое как смартфон, для выявления одной или нескольких уязвимостей. Затем исследователь делится информацией с владельцем сайта или изготовителем устройства, чтобы поспособствовать повышению безопасности. По прошествии достаточного для устранения уязвимостей времени исследователь публикует информацию о ней. Этот процесс называется **белым хакерством** с отсылкой к ранним американским вестернам, в которых главные герои носили белые шляпы, а злодеи — черные.
- **Инсайдеры (сознательные и невольные).** Наиболее распространенным фактором, способствующим кибервторжениям в защищенные сети, является действие, предпринятое авторизованным пользователем компьютерной системы в этой сети. Во многих случаях это действие заключается в простом щелчке мышью по ссылке в электронном письме, которая указывает на интересующую получателя тему. Это может привести к установке вредоносного ПО, которое быстро распространяется по предположительно защищенной сети. В других случаях, например из-за недовольства условиями работы, пользователь намеренно предпринимает шаги, позволяющие злоумышленникам проникнуть в сеть. В любом случае злоумышленники взламывают сеть, получая точку опоры, которая дает им возможность и далее использовать предположительно безопасную сеть в своих целях. Как только злоумышленники достигнут этого уровня доступа, они могут предпринять любые действия, которые пожелают, включая извлечение конфиденциальной информации о компании или установку программ-вымогателей в критически важных компьютерных системах.

По состоянию на 2021 г. программы-вымогатели являются наиболее распространенным вредоносным ПО для частных лиц, предприятий и других организаций.

Методы кибератак

Независимо от факторов, мотивирующих кибер-злоумышленников, они используют несколько общих типов атак на компьютерные системы, будь то веб-сервер, промышленная система управления или ПК, принадлежащий отдельному лицу. Ниже перечислены некоторые из наиболее вероятных типов атак.

- **Фишинг.** Это слово обозначает основанные на использовании электронной почты попытки убедить получателя предпринять какие-либо действия, которые будут способствовать достижению целей злоумышленника. В электронном письме может содержаться просьба к получателю перейти по ссылке или, возможно, загрузить и открыть файл, а иногда просто ответить отправителю в случае интереса к теме исходного электронного письма. В большинстве случаев целью фишинга является побудить получателя к каким-либо дей-

ствиям, которые приведут к установке вредоносного ПО, предоставляющего отправителю контроль над компьютером получателя.

- **Ботнеты.** Ботнет (от англ. *robot network* — сеть роботов) или бот-сеть — это совокупность компьютеров, которые были заражены вредоносным ПО и находятся под управлением одного злоумышленника. Этого управляющего называют **бот-мастером**. Компьютерам, входящим в ботнет, можно поручать выполнение злонамеренных задач, таких как рассылка фишинговых электронных писем в попытке привлечь в ботнет больше участников или проведение атак типа **отказ в обслуживании** (denial of service, DoS), когда компьютеры — участники ботнета пытаются перегрузить веб-сайты одновременной отправкой множества запросов на обслуживание.
- **Атаки со взломом паролей.** В большинстве компьютерных систем пароли, которые пользователи вводят для входа в систему, хранятся не в том формате, в котором они вводятся с клавиатуры. Вместо этого пароли передаются с помощью криптографического алгоритма хеширования, который преобразует пароль в неузнаваемую строку двоичных данных.

  Важные особенности криптографического хеш-алгоритма заключаются в том, что ввод одного и того же пароля всегда приводит к получению одного и того же хеш-значения, но знание этого хеш-значения не дает злоумышленнику простого способа определить соответствующий пароль. Несмотря на то что обладание хеш-значением пароля не позволяет напрямую определить пароль, злоумышленники все же могут найти пароль пользователя по его хеш-значению. Стандартный подход к определению пароля пользователя при известном значении хеш-функции заключается в простом переборе всех возможных паролей, пока не будет найдено соответствующее хеш-значение. Этот метод называется **взломом пароля методом полного перебора**. Лучшей защитой от такого типа атак является использование длинных паролей (15 символов или более) со смешанным набором символов верхнего и нижнего регистра, а также некоторых специальных символов, таких как * и &. Короткие пароли (менее 10 символов) или пароли, которые можно найти в словаре, довольно легко взломать, как и такие, к сожалению, достаточно распространенные пароли, как `123456`, `qwerty` или `password1`.
- **Использование уязвимостей.** Исследователи систем безопасности (белые хакеры) и хакеры-злоумышленники (черные хакеры) прилагают огромные усилия для обнаружения уязвимостей кибербезопасности в существующих операционных системах, приложениях, веб-сайтах и мобильных устройствах. Белые хакеры преследуют цель повышения уровня кибербезопасности для всех. Сначала они информируют разработчика об уязвимой системе или приложении, а затем, по прошествии времени, необходимого для устранения уязвимости, публикуют информацию о ней для всеобщего обозрения. Белые хакеры руководствуются не только альтруистическими мотивами — обнаружение критической уязвимости в крупной программной системе может привести к известности, повышению репутации и уважению со стороны коллег-

исследователей. Черные хакеры используют любую уязвимость, которую они обнаруживают, для совершения преступных действий против компьютеров, которые восприимчивы к такой атаке. Часто только после того, как системы были атакованы с использованием ранее неизвестной уязвимости, разработчики систем и независимые исследователи (белые хакеры) анализируют вредоносное ПО-нарушитель и обнаруживают уязвимость в целевой системе.

Уязвимости классифицируются по времени (в днях), в течение которого системные администраторы и пользователи должны подготовить средства защиты от вновь обнаруженной уязвимости, прежде чем эта уязвимость будет использована для атаки. В худшем сценарии для владельцев и пользователей уязвимой компьютерной системы атаки могут начаться вообще без предупреждения. В этом случае уязвимость называют **уязвимостью нулевого дня** — это означает, что никакого предупреждения о нападении заранее не было.

Типы вредоносного программного обеспечения

Вредоносным называют программное обеспечение, установленное в компьютерной системе без разрешения владельца и пытающееся выполнять действия, нежелательные для владельца компьютера.

Вредоносное ПО (malware, MALicious softWARE) — это категория программ, которые выполняют нежелательные действия и часто мешают нормальной работе компьютера. Ниже перечислены некоторые из наиболее распространенных типов вредоносных программ.

- **Шпионское ПО** — это программное обеспечение, которое извлекает личную информацию о пользователе компьютера и другие ценные данные и передает их злоумышленнику. К такой информации относятся имена пользователей и пароли для учетных записей интернет-сервисов и адреса посещенных веб-сайтов. Шпионские программы могут использовать эту информацию для отображения всплывающих рекламных объявлений, ориентированных на предполагаемые интересы пользователя. Тактика шпионских программ аналогична методам, обычно применяемым в рекламе законопослушными компаниями, поэтому отнесение части очевидных шпионских программ к вредоносному ПО может оказаться непростой задачей.
- **Программы-вымогатели.** Как указывалось ранее в этой главе, программы-вымогатели чаще всего шифруют файлы данных пользователя, а затем требуют оплаты в обмен на предоставление ключа для расшифровки данных. Другой формой вымогательства с помощью программ является кража личной информации физического лица или конфиденциальной информации компании и требование оплаты для предотвращения ее публичного раскрытия. Атаки программ-вымогателей часто нацелены на предприятия и организации, такие как больницы. Использующие эти методы злоумышленники обычно требуют оплаты в криптовалюте, которую, по их мнению, невозможно отследить. Несмотря на то что во многих случаях выплата выкупа приводит к ус-

пешной расшифровке зашифрованных данных или отказу от публичного распространения частной информации, нет никаких гарантий, что такая выплата приведет к благоприятному для жертвы исходу.

- **Вирусы.** Подобно биологическому вирусу, программный вирус размножается, заражая другие компьютеры собственными копиями. Помимо выполнения функций шпионского ПО или программы-вымогателя либо реализации кибератаки в какой-либо иной форме, вирус содержит код, который пытается получить по сети доступ к другим компьютерным системам и установить на них свои копии. Вирус внедряется в существующую компьютерную программу путем внесения в нее изменений для добавления вирусного кода. После этого измененная программа становится носителем вируса и предпринимает попытки заразить другие компьютеры. Программу, которая была изменена путем включения в нее вируса, называют зараженной вирусом. Требование репликации через программное приложение-носитель отличает компьютерный вирус от червя, которому для этой цели приложение-носитель не требуется.
- **Черви.** Подобно вирусу, компьютерный червь представляет собой автономную программу, которая пытается размножиться путем установки своих копий на другие компьютеры, доступные по сети. Червю, в отличие от компьютерного вируса, не требуется зараженное приложение-носитель. Как и вирус, червь может содержать код, который действует как шпионское ПО или программа-вымогатель или реализует атаку какого-либо другого типа.
- **"Человек посередине" (man in the middle, MITM).** В атаках этого типа злоумышленник пытается разместить программное обеспечение на пути обмена данными между двумя взаимодействующими приложениями или компьютерами. В случае успеха такая атака позволяет злоумышленнику перехватить конфиденциальную информацию, например имена пользователей и пароли, а также изменять данные по мере их передачи между взаимодействующими узлами.
- **Отказ в обслуживании (denial of service, DoS).** При атаке этого типа целевая система, например веб-сервер, подвергается бомбардировке огромным количеством ложных запросов. Цель атаки — сделать целевую систему недоступной для использования законными пользователями в течение длительного периода. DoS-атаки популярны среди хактивистов, которые применяют этот подход для создания проблем и привлечения негативного внимания к предприятиям и другим организациям, которые являются объектами их гнева.
- **Внедрение SQL-кода.** Язык программирования баз данных **Structured Query Language** (**SQL**) часто используют в веб-приложениях для связи между пользовательским интерфейсом и базами данных, содержащими такую информацию, как учетные данные пользователей или сведения о продуктах для продажи через веб-сайт. Если разработчики веб-приложения проявят недостаточную осторожность в применении методов безопасного программирования, пользователи приложения могут получить возможность формировать входные

данные, воспринимаемые SQL-интерпретатором как исполняемый код. В случае успеха эта уязвимость позволяет злоумышленнику извлекать данные из базы данных и изменять их, а иногда приводит к более разрушительной атаке на сеть, в которой находится база данных.

- **Перехват нажатий клавиш. Клавиатурные шпионы** относятся к типу вредоносного ПО, которое записывает последовательность нажимаемых пользователем компьютера клавиш и пересылает эту информацию злоумышленнику. Клавиатурный шпион может записывать важную личную информацию, такую как имена пользователей и пароли для банковских счетов и счетов кредитных карт. В более общем плане вредоносные программы этого типа могут выполнять такие функции, как сохранение снимков экрана с дисплея компьютера-жертвы и даже запись с видеокамеры и микрофона, подключенных к зараженному компьютеру или смартфону. Функции клавиатурного шпиона часто являются составной частью более крупного вредоносного ПО, содержащего шпионские программы, программы-вымогатели или вирусы.
- **Атаки на объекты инфраструктуры.** Управление работой многих объектов крупномасштабной инфраструктуры, таких как электростанции, линии электропередачи, плотины, нефтеперерабатывающие заводы и трубопроводы, основано на компьютеризированных системах управления. По сложившейся традиции специализированные компьютеры, используемые для этих целей, обладали лишь самыми элементарными функциями безопасности. Специалисты по кибербезопасности очень обеспокоены тем, что тщательно продуманные атаки на эти системы с помощью вредоносных программ могут привести к серьезным последствиям, таким как массовые отключения электроэнергии или взрывы на нефтеперерабатывающих заводах.

Вредоносные программы некоторых типов сочетают в себе несколько подобных возможностей и способны выполнять такие действия, как проведение тщательного анализа только что зараженной системы, прежде чем принять решение о продолжении атаки. Предметом этого анализа могут быть такие факторы, как страна, в которой находится зараженная система, компания, которой принадлежит система, и конкретные приложения, установленные на компьютере.

Некоторые вредоносные программы попадают в компьютер в зашифрованном виде и выполняют расшифровку только по мере подготовки отдельных сегментов кода к выполнению. Цель шифрования кода состоит в том, чтобы максимально затруднить исследователям кибербезопасности изучение кода и понимание его назначения. Шифрование также может помочь вредоносному коду преодолеть автоматическую защиту, обеспечиваемую антивирусным программным обеспечением.

Действия после проникновения

После того как злоумышленник получает удаленный доступ к целевой системе, перед ним открывается множество вариантов дальнейшего использования этой системы, а также других компьютеров в той же сети. Одним из первых шагов, которые

злоумышленник обычно предпринимает после получения доступа к системе-жертве, является установка программ, обеспечивающих непрерывный доступ, даже если текущий авторизованный пользователь выходит из системы или система перезагружается. При атаках многих типов этот шаг выполняется автоматически, как часть последовательности действий по первоначальному проникновению в компьютер-жертву. В других случаях злоумышленник, проводящий удаленное исследование целевой системы, устанавливает этот код сразу после получения доступа.

Первый вход злоумышленника в систему-жертву часто происходит на уровне привилегий пользователя, который выполнил действие, открывшее доступ, например щелкнул по ссылке, откуда была загружена и установлена вредоносная программа. Многие виды вредоносной деятельности требуют лишь ограниченного уровня привилегий обычного пользователя. Примерами этого являются программы-вымогатели, которые шифруют файлы данных пользователя, или ботнеты, которые бомбардируют целевой веб-сервер запросами, используя ограниченные привилегии, доступные обычному пользователю.

Атаки некоторых типов, например попытки получить доступ к защищенным сведениям в базе данных, содержащей информацию о клиентах, требуют привилегий уровня администратора. Следующим шагом после получения доступа к компьютерной системе на уровне непривилегированного пользователя и установки программы, обеспечивающей непрерывную связь между злоумышленником и системой-жертвой, может быть попытка перехода на более высокий уровень привилегий.

Повышение уровня привилегий — это процесс, который злоумышленник использует для получения более высокого уровня привилегий в целевой системе, чтобы обеспечить себе доступ к системным ресурсам, таким как файлы и базы данных, которые недоступны обычным пользователям.

Повышение уровня привилегий может быть достигнуто различными способами. В операционных системах, драйверах или приложениях, которые выполняются с более высокими уровнями привилегий, часто имеются известные уязвимости. Если система не обновляется регулярно путем установки пакетов исправлений известных ошибок, злоумышленники могут использовать эти уязвимости, чтобы повысить уровень своих привилегий и получить доступ к защищенным системным ресурсам.

Обеспечив постоянное подключение к системе-жертве и обзаведясь привилегиями администратора, злоумышленник получает полный контроль над компьютером. Он может извлечь любые данные, хранящиеся на компьютере, и установить любое программное обеспечение, которое пожелает. В хакерской терминологии компьютер, находящийся под полным административным контролем удаленного злоумышленника, считается *захваченным* (owned) этим злоумышленником. Если злоумышленник достиг такого уровня контроля незаметно для законных пользователей или программного обеспечения, такого как антивирусные инструменты, то на хакерском сленге такая атака называется *совершенным захватом* (perfect own), сокращенно *pwn* (произносится как "поун").

Одно из ограничений, о котором злоумышленник обычно должен помнить, — это необходимость избегать того, чтобы факт заражения компьютера вредоносным ПО

был слишком очевиден для его законных пользователей. Большинство пользователей не слишком беспокоит, если их компьютер работает немного медленнее, чем обычно, но появление на экране окна, в котором отображаются вводимые злоумышленником команды, позволит любому догадаться о том, что система была взломана. Изощренные злоумышленники стараются избегать действий, которые выдают их присутствие жертвам атак.

В этом разделе представлен обзор некоторых наиболее распространенных категорий и методов реализации угроз кибербезопасности. Приведенный список методов и типов вредоносных программ не является исчерпывающим, кроме того, постоянно разрабатываются новые способы атак на цифровые системы и устройства. Для архитектора компьютерных систем недостаточно понимать типы кибератак, которые были распространены в прошлом. Крайне важно иметь представление обо всем спектре атак, включая теоретически возможные, даже если некоторые из них ранее не наблюдались. В следующем разделе будет рассмотрен ряд ключевых функций, которые должны быть реализованы в вычислительных устройствах, чтобы обеспечить поддержание высокого уровня безопасности системы и ее интерфейсов с другими сетями, пользователями и устройствами в течение всего срока службы.

Особенности защищенного оборудования

В начале проектирования нового компьютера или цифрового устройства или при пересмотре проекта существующей системы крайне важно, чтобы разработчик компьютерной архитектуры рассматривал безопасность как требование высшего уровня. Даже самые элементарные решения в процессе разработки, такие как выбор модели процессора, скорее всего окажут заметное влияние на безопасность конечного изделия. На первом этапе этого процесса необходимо понять, какие типы данных и другой информации, связанной с технологиями, должны быть защищены от раскрытия лицам, не имеющим надлежащих полномочий.

Определите, что нуждается в защите

Ниже перечислены некоторые типы информации, хранящейся на компьютерах и в сетях, которая обычно нуждается в защите от несанкционированного раскрытия:

- личные данные, такие как пароли, номера социального страхования, финансовые данные и история болезни;
- конфиденциальная информация, принадлежащая предприятию, включая списки клиентов, данные о конструкции изделий и стратегические планы;
- запатентованные технологии, например конструкция цифровых схем в смартфоне;
- правительственная информация, например информация о национальной обороне и разведывательные данные, собранные правоохранительными органами.

При проектировании конкретной системы архитектор компьютеров должен постоянно помнить о типах информации, которую необходимо защитить. Это касается цифровых данных, содержащихся в системе, а также аппаратных средств, которые могут содержать уязвимости, предоставляющие злоумышленникам доступ к устройству.

Анализ безопасности системы должен включать в себя оценку уровня физического доступа к системе со стороны потенциальных злоумышленников. Что касается оборудования, предоставляемого в распоряжение конечных пользователей, такого как персональные компьютеры и смартфоны, пользователи могут совершать любые желаемые действия с этим оборудованием, включая его разборку и изучение компонентов под микроскопом.

Для оборудования, рассчитанного на работу в контролируемой среде, такой как ферма облачных серверов, защита от прямых физических атак может представлять меньшую проблему. Однако, учитывая возможность атак со стороны злонамеренных инсайдеров, необходимо подумать о том, чтобы как минимум обнаруживать случаи недопустимого вмешательства, даже если такое вмешательство является результатом действий сотрудников с благими намерениями.

Особой категорией интеллектуальной собственности, заслуживающей защиты, являются встроенное ПО и программный код в устройствах, поставляемых пользователям. Если производитель желает сохранить этот код в коммерческой тайне, процесс разработки должен включать действия по защите кода и предотвращению его раскрытия даже самыми умелыми и способными злоумышленниками.

Рассматривайте все типы атак

При проектировании функций безопасности цифровой системы важно учитывать не только типы атак, которые наблюдались в прошлом.

Архитектор должен широко оценивать ситуацию для определения категорий атак, которые еще не были замечены, но могут быть осуществлены, хотя бы теоретически. Такое мышление может привести к выявлению технологий, которые, вероятнее всего, не представляют особой угрозы в ближайшей перспективе (например, квантовые вычисления), но тем не менее могут быть эффективно нейтрализованы при приемлемых затратах ресурсов.

Вычислительную среду компьютерных систем, которым не требуется доступ к Интернету или к другим внешним сетям, обычно конструируют таким образом, чтобы обеспечить изоляцию от соединений с внешними сетями. Такую конфигурацию называют **изолированной**. Это означает, что между компьютерным оборудованием и любым потенциальным подключением к внешним сетям существует физическое разделение.

Построение физически изолированных вычислительных сред теоретически обеспечивает существенное повышение безопасности, однако в реальности преимущества этой архитектуры оказались ограничены. Для того чтобы любая компьютерная система сохраняла существенную ценность, обычно необходимо регулярно передавать в эту систему информацию в виде обновлений программного обеспечения и ис-

пользуемых на компьютере данных. Для изолированных систем эти обновления обычно поставляются на оптических дисках или портативных дисковых накопителях. Несмотря на благие намерения и усилия операторов систем, направленные на обеспечение безопасности, такой процесс передачи данных предоставляет удобную возможность для проникновения вредоносного ПО в защищаемую систему, а после заражения позволяет вредоносному ПО предпринимать попытки передачи данных из системы злоумышленникам и обратно, используя те же механизмы передачи.

Продвинутые хакеры постоянно работают над развитием методов атак на изолированные компьютерные системы и другие вычислительные среды с высоким уровнем безопасности. Разработчики защищенных вычислительных систем должны учитывать все возможности, которые злоумышленники могут использовать в своих попытках получить доступ к системе и эксплуатировать ее. Для создания надежной архитектуры крайне важен творческий подход.

Все возможные формы доступа к информации и пути утечки данных являются законной добычей для преданного своему делу злоумышленника. Даже если в компьютерной системе не предусмотрена возможность подключения к доступной извне сети, для решительного противника могут оказаться жизнеспособными другие потенциально экзотические атаки. Кибератаки, позволяющие передавать данные между процессами, в которых не предусмотрена возможность обмена данными, называются **атаками через скрытые каналы**. Ниже перечислены некоторые удивительные типы атак, которые продемонстрировали по крайней мере некоторую степень успеха.

- **"Простукивание строк".** Было показано, что современные устройства памяти DRAM, рассмотренные в предыдущих главах, уязвимы для атак, называемых **простукиванием строк** (row hammer). Слово "строка" в названии атаки относится к строкам битовых ячеек в устройстве памяти DRAM. Благодаря крошечному размеру каждой битовой ячейки и ее близости к соседним ячейкам в той же строке и в соседних строках при определенных условиях появляется возможность изменять состояние ("инвертировать биты") ячеек, находящихся в соседних строках.

  Для того чтобы вызвать этот эффект, код с высокой скоростью выполняет повторяющиеся обращения к целевой строке DRAM. Для того чтобы атака была успешной, выполняющий ее код должен гарантировать, что его запросы на доступ к памяти приводят к кеш-промахам, при которых задействуются внутренние схемы DRAM. Атаки типа "простукивание строк" продемонстрировали возможность добиться повышения уровня привилегий на компьютерах, основанных на архитектуре *x*86.

- **Колебания энергопотребления.** Даже если злоумышленникам удастся установить вредоносное ПО в изолированную компьютерную систему, извлечение собранных вредоносным ПО данных остается сложной задачей. Если невозможно вывести информацию через накопители, используемые для переноса данных в изолированную систему и обратно, должен быть найден альтернативный метод. Исследователи продемонстрировали, что вредоносное

ПО, запущенное в целевой компьютерной системе, может создавать колебания энергопотребления в электросети здания, которые можно отслеживать на линиях электропередачи за пределами здания. Тщательно закодировав цифровые данные в эти колебания, злоумышленники могут извлекать информацию из изолированного компьютера.

- **Колебания температуры.** Современные компьютеры оснащены вентиляторами, а иногда и системами жидкостного охлаждения, чтобы регулировать температуру системы и поддерживать ее в заданных пределах. Когда компьютеры расположены близко друг к другу, температура одного компьютера может влиять на показания температуры соседнего компьютера. Предположим, что вредоносное ПО установлено на двух близко расположенных компьютерах, один из которых находится в изолированной сети, а другой подключен к внешней сети. Было продемонстрировано, что преднамеренное изменение температуры на одном компьютере путем загрузки его процессора может вызывать измеримые изменения температуры на соседнем компьютере. Используя этот метод, вредоносное ПО на изолированном компьютере может передавать цифровые данные на компьютер, подключенный к внешней сети. Таким же образом можно передавать данные в противоположном направлении. Этот метод предлагает чрезвычайно малую скорость передачи данных, однако он позволяет передавать критически важные сведения, такие как ключи шифрования, которые могут быть использованы для обеспечения дальнейших атак.
- **Электромагнитное излучение.** Когда электрический ток протекает по проводнику, такому как USB-кабель, или по дорожке на печатной плате, в окружающую среду излучается электромагнитная волна. Если злоумышленник может разместить приемную антенну в пределах досягаемости этого сигнала, он получает возможность собирать передаваемую таким образом информацию. Конечно, в среде с высоким уровнем электрических помех, где одновременно работает множество компьютеров, может быть исключительно трудно извлечь сигнал, излучаемый какой-либо одной системой. Однако, если вредоносное ПО присутствует на целевом компьютере, оно может предпринять действия для преднамеренного генерирования электромагнитных колебаний по схеме, которая может быть обнаружена и декодирована подходящей системой приема. Случались и более странные вещи.

В этом разделе были представлены некоторые примеры экзотических атак, которым может подвергнуться критически важная компьютерная система. Этот список атак ни в коем случае не является полным, однако он содержит примеры угроз, которые системный архитектор должен учитывать при проектировании защищенной системы.

Особенности конструкции защищенных систем

Учитывая приведенные выше примеры возможных атак, мы можем выделить некоторые важные особенности, которые должны быть реализованы в аппаратных сред-

ствах защищенной компьютерной системы, чтобы обеспечить высокий уровень гарантий безопасности.

Надежное хранение ключей

Любые криптографические ключи, используемые системой для защиты данных, должны храниться таким образом, чтобы предотвратить их извлечение любыми мыслимыми способами. Обычно это означает, что ключи должны храниться в устройстве, например в чипе процессора, таким образом, чтобы предотвратить их извлечение любым программным методом. Это также означает принятие мер, пресекающих попытки извлечь ключи такими методами, как разборка интегральной схемы или использование сложных инструментов, таких как сканирующий электронный микроскоп.

Шифрование неактивных данных

Любые данные, хранящиеся в системе, должны быть защищены, когда питание системы выключено. Это означает, что даже если оборудование было разобрано и содержимое отдельных устройств памяти извлечено злоумышленником, данные внутри них должны оставаться в безопасности. Наиболее распространенным методом достижения такого уровня безопасности является шифрование данных с помощью ключа шифрования, который доступен для работы с данными в компьютере, но полностью защищен от раскрытия вне пределов его предполагаемого использования. Один из способов хранения такого ключа — в специальных регистрах процессора, доступ к которым закрыт даже для вредоносных программ, имеющих повышенные привилегии. Многие современные процессоры, включая крошечные встраиваемые устройства, начали оснащать тщательно продуманными криптографическими возможностями для этой цели.

Шифрование данных при передаче

Злоумышленник, имеющий физический доступ к каналу связи, может получить доступ к любым данным, передаваемым процессором или через интерфейс связи. Независимо от того, как передаются данные — по дорожке печатной платы или через глобальную сеть, любая конфиденциальная информация должна быть защищена на всем пути от ее источника до места назначения. Как и в предыдущем случае, общим подходом для реализации этой защиты является использование шифрования. Защита данных при передаче между двумя оконечными устройствами является более сложной задачей, чем шифрование и расшифровка данных в локальном хранилище, т. к. в этом случае шифрование и расшифровку необходимо выполнять в двух разных местах.

Концептуально, наиболее простым методом достижения этой цели является предоставление ключа шифрования системам на обоих концах канала связи. Однако доставка секретного ключа в обе системы затруднена, если между ними еще нет защищенного канала связи. Стандартный подход, практикуемый сегодня для создания безопасных каналов связи между системами, которые еще не имеют общего секретного ключа, заключается в применении шифрования с открытым ключом для

передачи секретного ключа с одного конца канала связи на другой. После этого секретный ключ используется для шифрования и расшифровки данных на каждом конце канала связи. Можно было бы просто применять процесс шифрования с открытым ключом для всех данных, совместно используемых системами, но оказывается, что шифрование с открытым ключом требует гораздо больших вычислительных затрат, чем работа с общим секретным ключом.

Генерирование криптостойких ключей

Когда требуется новый секретный ключ, например при создании безопасного канала связи между двумя компьютерными системами, крайне важно, чтобы этот ключ был абсолютно непредсказуем для любых внешних злоумышленников. Любой вновь созданный секретный ключ должен выглядеть совершенно случайным и абсолютно не связанным с любыми предыдущими или последующими ключами, созданными той же системой или другими системами. Традиционно самым простым способом создания на компьютере числа, имеющего случайный вид, было использование функций генерирования псевдослучайных чисел, доступных во многих библиотеках языков программирования. Однако оказалось, что последовательности чисел, создаваемые многими из этих алгоритмов, не дают результатов, похожих на случайные, и эти алгоритмические недостатки могут сделать задачу взлома алгоритмов шифрования значительно более простой, чем может показаться при первоначальном анализе. Современные криптографические генераторы случайных чисел используют специализированные аппаратные средства для создания *истинных* случайных чисел, предлагая возможности генерирования максимально защищенных криптографических ключей.

Процедура безопасной загрузки

В безопасной системе весь код, выполняемый на повышенном уровне привилегий, необходимо проверить на подлинность, прежде чем будет разрешено его выполнение. Это относится ко всему коду, выполняемому в процессе загрузки, а также к ядру и драйверам операционной системы. Стандартным подходом для решения этой задачи является прикрепление к каждому фрагменту кода цифровой подписи. Цифровая подпись содержит зашифрованное хеш-значение, которое вычисляется для всего блока исполняемого кода, независимо от того, хранится ли он в виде файла на диске или содержится во флеш-памяти процессора. Ключ, используемый для расшифровки хеш-значения цифровой подписи, должен храниться в аппаратных средствах процессора. При таком уровне защиты любая попытка заменить подлинный код на умышленно модифицированный потерпит неудачу, поскольку у злоумышленника нет необходимого ключа, чтобы сформировать действительную цифровую подпись для измененного кода.

Взломостойкая конструкция аппаратных средств

Особенности защищенных компьютерных архитектур, описанные в этом разделе, в той или иной степени зависят от аппаратных средств процессора, способных надежно хранить секретную информацию, такую как криптографические ключи. Для

обеспечения долгосрочной защиты конфиденциальной информации, обрабатываемой компьютером, архитектура аппаратных средств должна оставаться защищенной от любых вероятных атак. Ниже перечислены отдельные примеры методов, которые злоумышленник может использовать, чтобы попытаться извлечь конфиденциальную информацию, имея физический доступ к вычислительному устройству.

- **Физический доступ к устройству.** Выполнив такие действия, как аккуратное вскрытие корпуса интегральной схемы с помощью шлифовки или химической эрозии, злоумышленник может получить доступ к внутренним компонентам схемы. Получив такой доступ, можно будет электрическим способом зондировать компоненты схемы и извлекать из них информацию.
- **Мониторинг электромагнитного излучения.** Получив доступ к внутренним компонентам устройства, содержащего конфиденциальную информацию, можно использовать микроскопическую антенну для мониторинга активности определенных компонентов или информации, проходящей через внутренние соединения.
- **Микроскопическое исследование.** Для измерения распределения напряжения по поверхности схемы, например схемы массива памяти, можно использовать экзотические лабораторные приборы, такие как *сканирующий микроскоп напряжения*.

Применение некоторых из описанных в этом разделе методов атак против обычных компьютерных систем может показаться крайне маловероятным, однако вас может удивить то, насколько доступными могут быть эти методы. Хотя у большинства из нас нет собственного сканирующего микроскопа напряжения или доступа к такому устройству, имеются компании, которые владеют этими устройствами и готовы выполнить сканирование для клиентов за удивительно приемлемую цену. Это одна из причин, почему архитекторам компьютерных систем, которые должны содержать конфиденциальную или чрезвычайно ценную информацию, важно учитывать при их проектировании все возможные типы атак.

В следующем разделе мы обсудим конфиденциальные вычисления, где применяются строгие меры безопасности для защиты данных на протяжении всего жизненного цикла их обработки.

Конфиденциальные вычисления

Конфиденциальные вычисления — это недавняя разработка, целью которой является использование криптографии и средств защиты на аппаратном уровне для обеспечения постоянной защиты данных. Данные могут находиться в одном из трех состояний: в состоянии покоя, в движении или в обработке. **Данные в состоянии покоя** обычно хранятся в файлах на устройстве хранения данных. **Данные в движении** — это данные, перемещаемые через коммуникационную среду какого-либо типа. **Данные в обработке** — это данные, активно обрабатываемые процессором и находящиеся в основной памяти процессора.

Конфиденциальные вычисления направлены на обеспечение всесторонней защиты данных во всех трех вышеуказанных состояниях. Традиционные механизмы безопасности в каждый момент времени сосредоточены на одном состоянии, например на шифровании хранящихся на диске данных или на передаче информации на веб-сайт или с веб-сайта. Эти подходы пренебрегают необходимостью обеспечения такого же уровня защиты данных, находящихся в обработке.

Защита данных в обработке требует поддержки со стороны аппаратных средств процессора для изоляции приложений друг от друга и обеспечения защиты конфиденциального кода и данных. В качестве одного из примеров аппаратной поддержки конфиденциальных вычислений можно привести технологию Intel **Secure Guard Extensions (SGX)**, которая обеспечивает надежную изоляцию приложений и защиту данных в обработке.

Intel утверждает, что SGX защищает данные приложений даже тогда, когда BIOS, операционная система и само приложение были взломаны, а злоумышленник получил полный контроль над платформой.

Технология SGX создает изолированные области памяти, называемые анклавами. Каждый анклав содержит неадресуемые области памяти, которые содержат код и данные приложения на зашифрованных страницах памяти.

Приложения, созданные с использованием технологии SGX, состоят из двух частей: доверенной и общей. Общая часть приложения создает доверенную часть, которая содержит защищенный анклав. Анклавы считаются доверенными, поскольку после создания изменить их нельзя. Если анклав был изменен, это изменение будет обнаружено, и его выполнение будет запрещено. Код, запущенный в доверенной части приложения, получает доступ к данным внутри анклава в виде открытого (нешифрованного) текста. Любому коду, находящемуся за пределами анклава, такому как код из BIOS, операционной системы или даже из общей части того же приложения, запрещен доступ к области памяти анклава. Даже внешний по отношению к анклаву код, который выполняется на уровне привилегий ядра, не может получить доступ к защищенным данным внутри анклава.

Страницы памяти в защищенном анклаве могут быть вынесены во вторичное хранилище с использованием традиционных алгоритмов подкачки страниц. Когда защищенные страницы находятся в общем файле подкачки, шифрование кода и данных внутри страниц анклава защищает информацию, которую они содержат.

Технология SGX поддерживает концепцию аттестации программного обеспечения. Используя процедуру **аттестации программного обеспечения**, код, получающий удаленный доступ к функциям в защищенном анклаве, может проверить, что он взаимодействует с конкретным анклавом, с которым он намеревается работать, а не с подделкой. Процедура аттестации основана на обмене криптографическими цифровыми подписями для надежной проверки идентичности анклава.

Конфиденциальные вычисления особенно хорошо применимы в контексте удаленных вычислений. Термин **"удаленные вычисления"** описывает использование вычислительных ресурсов, которыми владеет и управляет недоверенная сторона, не являющаяся владельцем приложения.

Это описание применимо, например, к компании, которая использует поставщика облачных услуг для управления своими корпоративными вычислительными ресурсами. Удаленное приложение может работать с конфиденциальными данными в недоверенной вычислительной среде и поддерживать их безопасность на всех этапах работы.

Примером удаленных вычислений является коммерческий веб-сервер, запущенный в коммерческой облачной среде. Веб-приложение собирает и сохраняет конфиденциальную информацию, такую как данные кредитных карт клиентов веб-сайта. Используя технологию SGX и другие стандартные криптографические методы, можно зашифровать конфиденциальные данные каждого клиента на компьютере пользователя и перенести эти данные в защищенный анклав для обработки заказа. Для того чтобы обновить запись пользователя в базе данных приложения, конфиденциальная информация шифруется в защищенном анклаве перед ее передачей в базу данных для хранения. В каждый момент времени после того, как пользователь ввел конфиденциальную информацию в своем браузере, эта информация имеет криптографическую защиту и защищена от злоумышленников, которые могли проникнуть в недоверенную вычислительную среду.

Технология Intel SGX была выпущена в 2015 г. и присутствует в большинстве современных процессоров Intel. Для того чтобы использовать эту технологию, компьютерная система должна поддерживать базовую систему ввода вывода (BIOS), которая обеспечивает выполнение действий, необходимых для ее включения. Поддержка SGX широко распространена в представленных сегодня на рынке материнских платах и компьютерах, но это не означает, что она доступна автоматически во всех системах. Если вы решите, что для вашего приложения требуется SGX, вам необходимо убедиться, что процессор, материнская плата, BIOS и операционная система компьютера, который вы планируете использовать, поддерживают эту технологию.

К сожалению, как и во многих предыдущих попытках повысить уровень безопасности компьютерных систем с помощью аппаратных усовершенствований, исследователи безопасности выявили уязвимости в технологии SGX. Фактически, исследователи продемонстрировали возможность извлекать криптографические ключи и другую ценную информацию из приложений, работающих в защищенных анклавах SGX.

Техника, которую злоумышленники используют против SGX, основана на функциях упреждающего (или внеочередного) выполнения, реализованных в современных процессорах. Как обсуждалось в *главе 8*, упреждающее выполнение — это метод оптимизации, при котором процессор начинает выполнять код по обеим ветвям, предусмотренным в инструкции ветвления, до тех пор, пока не станет ясно, какую ветвь выбрал код. При упреждающем выполнении в кеш-памяти сохраняются данные, которые в конечном счете будут отброшены. Эти кешированные данные являются источником уязвимости.

Данный конкретный метод атаки на SGX называют **вводом загружаемого значения** (load value injection, LVI). Атака LVI позволяет не только считывать значения

из предположительно защищенного анклава, но и вводить значения данных в анклав. Intel определила действия, которые разработчики ПО могут предпринять для устранения угрозы LVI-атаки, однако эти меры могут оказать существенное влияние на производительность при выполнении программ. Intel также работает над обновлением микрокода процессора, которое устранит эту уязвимость.

Меры безопасности на уровне архитектуры

В процессе проектирования надежно защищенной компьютерной системы необходимо с самого начала учитывать широкий спектр требований к безопасности. Все аспекты проектирования системы, такие как выбор процессора и характеристик печатных плат, должны оцениваться в свете того, как эти компоненты могут улучшить или ухудшить общую безопасность системы.

Помимо обеспечения безопасности на самом низком уровне — уровне интегральных схем и печатных плат, также важно применять принципы безопасного проектирования и на других уровнях. Например, после выбора надлежащим образом защищенных цифровых компонентов и разработки ориентированной на безопасность топологии схемы цифрового устройства может потребоваться разработать взломостойкий корпус для печатной платы. Корпус может содержать прикрепленные к его поверхности проводники, позволяющие обнаружить попытки злоумышленника вырезать или просверлить в корпусе отверстие и тем самым получить доступ к внутренним компонентам. Этот подход часто используется в критически важных с финансовой точки зрения устройствах конечного пользователя, таких как считыватели кредитных карт в торговых точках.

Наилучшая защита системы обеспечивается применением методики безопасного проектирования на всех уровнях архитектуры системы как в аппаратных средствах, так и в программном обеспечении. В следующих разделах будут рассмотрены некоторые принципы проектирования, помогающие в создании безопасных систем.

Избегайте защиты посредством сокрытия информации

Один из заманчивых подходов, который годами использовался при разработке цифровых систем, заключается в том, чтобы как можно больше усложнить определение типа и назначения различных компонентов и соединений внутри цифрового устройства. Типичным способом в рамках этого подхода может быть механическое удаление обозначений с некоторых интегральных схем в устройстве, чтобы затруднить их идентификацию тому, кто его анализирует.

Другим способом скрыть функциональные возможности системы является нанесение на печатную плату части дорожек явно бессмысленным образом. Цель здесь состоит в том, чтобы максимально затруднить изучение устройства и обратную разработку его конструкции.

Термин **"обратная разработка"** уже давно используется для описания аналитического процесса, помогающего понять, как было сконструировано устройство или

ПО, без доступа к какой-либо документации, которая бы раскрывала процесс разработки. Существуют и законные причины для выполнения обратной разработки (например, при ремонте системы, для которой нет документации), но в данном случае мы сосредоточены на применении этого процесса в неблаговидных целях. Примером вредоносного применения обратной разработки могут послужить попытки получить доступ к такой конфиденциальной информации, как коммерческая тайна или защищенное авторским правом ПО, с преступными намерениями.

Исторически сложилось так, что самый большой недостаток подхода с преднамеренным сокрытием деталей устройства ценных цифровых систем заключался в том, что разработчики систем недооценивали возможности и находчивость тех, кто проводил обратную разработку. Похоже, что проектировщики, которые полагались на процесс внесения неясностей, считали, что так как они сами никогда не взялись бы за крайне утомительную работу, которая необходима для анализа намеренно вычурного и переусложненного устройства, никто другой не будет пытаться это сделать.

Известно много случаев, когда с помощью обратной разработки удавалось решать такие задачи, как идентификация отдельных сигналов, связанных с интерфейсом отладки, скрытым в сложной конструкции печатной платы. Заполучив информацию об этих соединениях, взломщики смогли подключить аппаратную систему отладки и извлечь весь защищенный код и другую информацию, содержащуюся в устройстве.

Среди специалистов по разработке различных систем распространено мнение, что только они могут разобраться в конструкции и поведении такого сложного устройства, а их понимание основано лишь на том, что у них есть доступ к системной документации, недоступной для злоумышленников. Это предположение часто является неверным по двум следующим причинам.

- Во-первых, многие взломщики, практикующие обратную разработку, очень умны и ориентированы на детали, что позволяет им методично сопоставлять функции и поведение системы на удивление полным образом.
- Во-вторых, было бы неразумно предполагать, что системная документация останется в безопасности и будет скрыта от злоумышленников в течение длительного времени. Кибератаки и промышленный шпионаж, направленные против высокотехнологичных компаний, широко распространены. Существует значительная вероятность того, что решительные злоумышленники в какой-то момент получат доступ к некоторой или всей системной документации. Эту вероятность следует учитывать при оценке использования подхода с внесением неясностей вместо применения проверенных механизмов безопасности для защиты критически важной информации.

Отказавшись от подхода, заключающегося в использовании непрозрачных процессов проектирования, чтобы помешать злоумышленникам взломать систему, мы должны обратиться к эффективным и проверенным временем подходам безопасного проектирования, от которых можно ожидать надежной работы как сейчас, так и в будущем.

Комплексный подход к безопасному проектированию

Используя концепции, рассмотренные ранее в этой главе, при проектировании надежно защищенной компьютерной системы необходимо применять комплексный подход, начиная с ее базовых компонентов и заканчивая конечными деталями. Это позволит обеспечить высочайший уровень безопасности. Там, где это осуществимо, следует использовать возможность математически доказать, что система безопасна по своему замыслу. В то же время доказать безопасность архитектуры системы во всех ее аспектах невозможно (поскольку даже для самого простого приложения обычно нельзя продемонстрировать безопасность математически), однако это можно попытаться сделать в отношении наиболее критических аспектов устройства системы, таких как процесс входа пользователя в систему.

Определив все точки доступа, через которые киберпреступник может попытаться взломать систему, и показав реализацию комплексных мер безопасности в каждой из этих точек, разработчики могут продемонстрировать определенный уровень уверенности в защищенности системы от атак. Если требуется обеспечить высочайший уровень безопасности системы, этот анализ должен охватывать все возможные методы взлома, описанные ранее в этой главе, включая те, которые считаются наименее вероятными.

Помимо учета соображений безопасности во всех аспектах конструкции аппаратных средств и программного обеспечения системы, важно обеспечить, чтобы пользователи системы работали с минимальным уровнем привилегий, который им необходим для выполнения своих должностных обязанностей. Это тема следующего раздела.

Принцип наименьших привилегий

Некоторым пользователям требуется привилегированный доступ для управления и обслуживания защищенной компьютерной системы, тогда как многим другим пользователям для выполнения своей работы требуются лишь минимальные привилегии обычного пользователя. *Принцип наименьших привилегий* обобщает идею о том, что привилегии каждого пользователя в компьютерной системе не должны превышать уровень, необходимый для выполнения его должностных обязанностей. Например, пользователь, которому требуется изучать и обновлять информацию в корпоративной базе данных, должен быть наделен привилегиями, необходимыми для выполнения этих задач, но он не должен иметь дополнительных привилегий, которые, например, требуются для системного администрирования, не входящего в круг его обязанностей.

Принцип наименьших привилегий гарантирует, что пользователи имеют разрешения и права доступа, необходимые им для выполнения своей работы, но не более того.

Когда пользователя переводят на другую работу или он увольняется, для организации, заботящейся о безопасности, особенно важно скорректировать права доступа сотрудника сразу после изменения его обязанностей, чтобы гарантировать, что этот

сотрудник (который может быть недоволен, если его наказали или уволили) больше не имеет доступа к информации, не связанной с его работой.

Для эффективного применения принципа наименьших привилегий требуется нечто большее, чем тщательный процесс проектирования с предоставлением минимальных привилегий пользователям и приложениям во время разработки системы. Системные администраторы и операторы должны на постоянной основе контролировать, чтобы всем новым приложениям и пользователям предоставлялись лишь минимальные уровни привилегий, необходимые для выполнения назначенных им функций.

Архитектура нулевого доверия

Традиционный подход к проектированию защищенной компьютерной системы, описанный в предыдущих разделах, основан на многоуровневой модели безопасности, цель которой заключается в том, чтобы даже в случае сбоя определенной функции безопасности оставшихся уровней было достаточно для сохранения общей безопасности.

Учитывая регулярное появление публикаций о широком спектре успешных атак, демонстрирующих полный провал предпринятых мер безопасности в системах, которым доверяли защиту критически важных личных, коммерческих и правительственных данных, этот подход явно имеет некоторые ограничения.

В традиционной модели безопасности границей зоны безопасности считается внешний периметр сети. Внутри этого уровня защиты обмен данными по внутренней сети считается доверенным, а компьютерным системам и пользователям предоставляются многие типы доступа.

В архитектурной модели нулевого доверия любое сообщение, которое поступает в компьютерную систему, содержащую защищенную информацию, должно рассматриваться как потенциально вредоносное. Запрос обрабатывается только после установления подлинности его источника и подтверждения допустимости запрошенного действия.

Агентство национальной безопасности (АНБ) США определило следующий набор руководящих принципов, которые необходимо применять при разработке компьютерной архитектуры с нулевым доверием.

- **"Никогда не доверяй и всегда проверяй".** Каждая компьютерная система, пользователь, сетевое устройство или другой источник данных в сети всегда должны рассматриваться как ненадежные. Каждый получатель пользовательских данных или другой потребитель данных должен явно устанавливать подлинность каждой части полученных входных данных. Кроме того, при предоставлении привилегий пользователям или приложениям следует придерживаться принципа наименьших привилегий, который диктует, что предоставляются только минимально необходимые привилегии.
- **"Предполагай худшее".** Всегда предполагайте, что в сеть проникли враждебные субъекты, и действуйте соответствующим образом. Это означает, что

ответом по умолчанию на любой ввод данных или запрос обслуживания является игнорирование или отклонение запроса. Ответ возможен только после тщательной проверки источника и подтверждения допустимости запрошенного действия. Системные администраторы и автоматизированные средства должны постоянно отслеживать все аспекты конфигурации сети, выявлять попытки несанкционированного доступа или изменения конфигурации и оперативно принимать ответные меры, чтобы остановить любую вредоносную активность и восстановить одобренную рабочую конфигурацию. Все уместные действия пользователей, приложений и сетевой инфраструктуры должны регистрироваться и проверяться, чтобы быстро обнаружить любое отклонение от одобренных операций.

- **"Проверяй явно".** Каждая попытка доступа пользователя или приложения к защищенным ресурсам требует отдельной операции проверки. Процесс проверки должен затрагивать множество атрибутов запрашивающего пользователя или приложения для надежного установления подлинности источника запроса. Например, система может потребовать использования двухфакторной аутентификации, когда пользователь вставляет криптографически защищенную карту доступа в считывающее устройство, а затем вводит PIN-код. Такая карта и PIN-код представляют собой два независимых атрибута, которые обеспечивают аутентификацию в качестве доверенного пользователя.

Для полной реализации архитектуры нулевого доверия требуется внедрение механизма принятия решений, который оценивает решения о предоставлении доступа в контексте всей имеющейся информации об отправителе запроса и пункте назначения запроса. Помимо аутентификации отправителя запроса и проверки применения наименьшего уровня привилегий, механизм принятия решений должен учитывать любую дополнительную информацию, которая может указывать на повышенные риски, связанные с запросом. Дальнейшая обработка запроса пользователя разрешается только после проверки всей доступной информации и определения того, что уровень риска, связанного с одобрением запроса, не превышает заранее установленного порогового значения.

В следующем разделе будут рассмотрены некоторые способы, благодаря которым слабые места в защите программного обеспечения, работающего на надлежащим образом защищенном оборудовании, могут привести к появлению уязвимостей в компьютерной системе.

Обеспечение безопасности системного и прикладного ПО

При разработке аппаратного уровня защищенной системы важно придерживаться безопасного и проверенного подхода к проектированию на всех уровнях архитектуры программного обеспечения. Далее мы рассмотрим некоторые из способов, с помощью которых программный код может вносить уязвимости в защищенные компьютерные системы.

Общие слабые места программного обеспечения

В этом разделе перечислены несколько категорий слабых мест программного обеспечения, которые традиционно вызывали серьезные проблемы с безопасностью в операционных системах, приложениях и веб-серверах. Эти уязвимости иногда возникают из-за того, что разработчики ПО делают предположения о поведении пользователя, которые оказываются неверными. В других случаях разработчики просто не осознают, что следование определенным шаблонам программного обеспечения ведет к созданию небезопасных конструкций.

Некоторые из перечисленных далее методов, вероятно, применимы лишь в конкретных языках программирования, но разработчики ПО должны знать, что небезопасный код можно создать на любом языке программирования.

Вот список некоторых наиболее распространенных слабых мест программного обеспечения, которые наблюдались в последние годы.

Переполнение буфера

Переполнение буфера происходит, когда набор входных данных превышает объем памяти, выделенной для ввода, и избыточные данные перезаписывают участки памяти, которые могут содержать другие важные данные. Это традиционно было проблемой с программами, написанными на языках программирования C и C++. В типичном сценарии программа предлагает пользователю ввести текстовую строку, которая, как ожидается, будет короткой, например имя пользователя.

Для приема входных данных разработчик может выделить буфер, например на 80 символов, предполагая, что длина имени пользователя никогда не превысит это значение. Если код, обрабатывающий входные данные, принимает более 80 символов и сохраняет их в памяти, данные по адресам за пределами 80-символьного буфера будут перезаписаны. В традиционных реализациях языка C входной буфер, скорее всего, будет расположен в стеке процессора. Путем ввода специально сконструированной строки, злоумышленник может вставить нужный ему код, содержащий инструкции процессора, а также перезаписать адрес возврата для текущей функции, который хранится в стеке. Подменив адрес возврата адресом вредоносного кода, злоумышленник может выполнить любые инструкции, которые он пожелает, начиная с момента возврата из функции ввода. Первоначальной целью злоумышленников обычно является попытка получить удаленный доступ к командной строке (также называемой **командной оболочкой**) системы-жертвы. По этой причине вводимый код, который выполняет взлом этого типа, вызывают **кодом оболочки**.

Межсайтовое выполнение сценариев

Межсайтовое выполнение сценариев — это уязвимость, характерная для веб-приложений. При атаке с ее использованием злоумышленник находит способ внедрения вредоносных исполняемых сценариев на веб-страницы, которые затем дос-

тавляются с заслуживающего доверия сайта его пользователям. Эти сценарии выполняются в контексте браузера жертвы и могут предпринимать такие действия, как пересылка злоумышленнику учетных данных жертвы для входа в систему (например, файла cookie для авторизации). Благодаря этому злоумышленник может выдать себя за жертву на заслуживающем доверия веб-сайте и получить доступ к личной информации жертвы. Разработчикам программного обеспечения доступно несколько механизмов для проверки и очистки входных данных пользователей веб-сайта, которые эффективно предотвращают межсайтовое выполнение сценариев. К сожалению, не все веб-разработчики в полной мере используют эти функции безопасности, поэтому эта уязвимость продолжает существовать на многих веб-сайтах.

Внедрение SQL-кода

Как обсуждалось ранее в этой главе, многие веб-приложения используют базу данных для хранения пользовательской информации, такой как имена пользователей, хешированные пароли и содержимое сайта, например сообщения пользователей и загруженные изображения. Многие сайты для работы с информацией в базе данных и добавления новых данных, вводимых пользователями, используют язык SQL. Потенциальная уязвимость возникает, если пользователь намеренно вводит на сайте данные, которые могут быть интерпретированы как код SQL, и сайту не удается очистить входные данные таким образом, чтобы предотвратить выполнение введенного пользователем кода.

Например, этот серверный код извлекает имя пользователя, введенное в поле на веб-странице после того, как пользователь нажимает кнопку **Отправить**.

```
txtUserName = getRequestString("UserName");
```

На следующем этапе обработки простая реализация серверного кода может создать команду в синтаксисе SQL, которая извлечет из базы данных запись, связанную с введенным именем пользователя:

```
txtSqlCmd = 'SELECT * FROM Users where UserName = "' + txtUserName + '"';
```

Например, если пользователь ввел в качестве имени пользователя `Алиса`, текстовая строка, содержащая результирующую команду SQL, будет следующей:

```
SELECT * FROM Users where UserName = "Алиса"
```

Затем сервер передает эту команду интерпретатору команд базы данных, и, если было указано действительное имя пользователя, будет возвращена запись данных пользователя.

Проблема с этим подходом заключается в использовании текстовой строки, введенной пользователем непосредственно в командной строке SQL. Злоумышленник может использовать синтаксис SQL для изменения поведения операции доступа к

базе данных. Например, вместо ввода действительного имени пользователя злоумышленник может ввести следующую текстовую строку:

```
" or ""="
```

Когда эта строка предоставляется в качестве имени пользователя, выполняется следующая команда SQL:

```
SELECT * FROM Users where UserName = "" or ""=""
```

Эта команда указывает базе данных вернуть все записи, где имя пользователя является пустой строкой (что крайне маловероятно) или где пустая строка равна пустой строке. Поскольку пустая строка всегда равна пустой строке, условие будет истинным для всех записей, и база данных вернет все содержащиеся в ней пользовательские записи. Если серверный код затем представит результаты выполнения этой SQL-команды браузеру пользователя, злоумышленнику удастся извлечь всю пользовательскую базу данных этого веб-сайта.

Этот сценарий может показаться малопонятным методом атаки на веб-сайт, но вы должны понимать, что многие веб-сайты, часто принадлежащие известным и высоко оцениваемым компаниям, становятся жертвами разрушительных атак, очень похожих на описанную здесь.

Обход каталога

Уязвимость с обходом каталога возникает, когда сетевое приложение, обычно веб-сервер, непреднамеренно предоставляет своим пользователям некоторую степень доступа к своей структуре каталогов. Обычно операторы сайта предоставляют пользователям доступ к подкаталогам ниже основного каталога приложения для извлечения данных, хранящихся в этих каталогах. Данная уязвимость может возникнуть, если логика веб-сервера позволяет пользователям переходить *вверх* на один или несколько уровней каталогов, использующих этот метод.

В операционных системах Windows и Linux элемент пути к каталогу, состоящий из двух последовательных точек, означает переход в каталог более высокого уровня. Например, следующий URL-адрес показывает, как попытаться получить файл, содержащий зашифрованные пароли, в системе Linux, работающей под управлением стандартного веб-сервера:

```
http://www.example.com/../../../../etc/shadow
```

В случае успеха посещение этого URL-адреса с помощью браузера позволяет злоумышленнику получить хешированные версии всех паролей в системе. Затем злоумышленник может использовать методы взлома паролей методом полного перебора, чтобы попытаться восстановить пароли пользователей.

В этом разделе перечислены лишь некоторые из наиболее распространенных уязвимостей программного обеспечения, которые исторически использовались для нанесения серьезного ущерба компаниям и частным лицам, чья личная информация хранится в компьютерных системах.

База данных под названием **Общий каталог уязвимостей** (common weakness enumeration, CWE), доступная по адресу **https://cwe.mitre.org/index.html**, содержит список слабых мест программного обеспечения и аппаратных средств, составленный на основе данных, предоставленных пользователями по всему миру. В частности, список из 25 наиболее опасных уязвимостей программного обеспечения доступен по адресу **https://cwe.mitre.org/data/definitions/1337.html**. Этот список содержит широкий обзор наблюдаемых в настоящее время уязвимостей программного обеспечения, вызывающих серьезные проблемы безопасности.

Проверка безопасности исходного кода

Одним из способов быстро получить информацию о слабых местах имеющегося ПО является применение инструмента автоматической проверки безопасности исходного кода для оценки кодовой базы и классификации выявленных в ней проблем по степени серьезности.

В зависимости от языка или языков программирования, используемых для написания кода, можно найти бесплатные инструменты для выполнения такого сканирования. При выборе бесплатного сканера исходного кода, который вы планируете применить, обязательно ознакомьтесь с отзывами пользователей и убедитесь в том, что это действительно полезный инструмент. Для некоторых языков программирования может потребоваться приобретение инструмента проверки безопасности, который может оказаться довольно дорогим.

Инструменты автоматической проверки безопасности могут выявлять многие категории проблем в исходном коде, включая перечисленные выше слабые места, а также другие дефекты, указанные в базе данных CWE. Эти инструменты также выявляют другие проблемы в коде, такие как наличие сомнительных функций и использование конструкций, снижающих производительность.

Вместо того чтобы пытаться перечислить эти инструменты здесь, я предлагаю вам воспользоваться поиском в Интернете с целью найти инструменты автоматической проверки безопасности, ориентированные на языки, которые вы используете для своих приложений.

Резюме

В этой главе были представлены вычислительные архитектуры, подходящие для решения задач, требующих исключительных гарантий безопасности. Такой высокий уровень защиты востребован в критически важных областях, включая системы национальной безопасности и обработку финансовых транзакций. Эти системы

должны быть устойчивы к широкому спектру угроз кибербезопасности, в том числе к проникновению вредоносного кода, атакам через скрытые каналы и путем физического доступа к аппаратным средствам компьютеров. Темы, рассмотренные в этой главе, включали угрозы кибербезопасности, шифрование, цифровые подписи и архитектурные решения для защиты аппаратных средств и программного обеспечения.

Прочитав эту главу, вы научились определять категории угроз кибербезопасности, с которыми может столкнуться система, и разбираться в способах обеспечения безопасности аппаратных средств современных компьютеров. Теперь вы знаете, как можно избежать брешей в безопасности системных архитектур и как защищенная компьютерная архитектура может помочь обеспечить безопасность в программных приложениях.

В следующей главе мы представим концепции, связанные с блокчейном, открытым, криптографически защищенным реестром, содержащим последовательность транзакций. В ней дается общий обзор процесса майнинга биткоинов и обсуждаются аппаратные архитектуры компьютерных систем, подходящие для реализации этого процесса.

Упражнения

1. Для всех своих учетных записей в Интернете, содержащих важные данные, настройте двухфакторную аутентификацию (где она поддерживается). К таким учетным записям относятся банковские счета, электронная почта, социальные сети, хранилища кода (если вы разработчик программного обеспечения), медицинские услуги и все остальное, что для вас ценно. На всех этапах этого процесса необходимо заручиться гарантией того, что вы используете информацию и программные приложения исключительно из надежных источников.
2. Для тех своих учетных записей в Интернете, которые содержат ценную информацию, но могут быть защищены двухфакторной аутентификацией, создайте надежные пароли. Надежный пароль должен быть длинным (15 символов или более) и содержать прописные и строчные буквы, цифры и специальные символы (например, `! " # $ % & ' ( ) * +`). Для того чтобы отслеживать эти сложные пароли, установите и используйте надежное приложение для хранения паролей. Будьте осторожны при выборе такого приложения и тщательно выбирайте его источник.
3. Обновите операционную систему, а также другие приложения и сервисы (например, Java) на всех компьютерах и других устройствах, находящихся под вашим контролем. Это послужит гарантией, что новые функции безопасности, включенные в эти обновления, начнут работать для вашей защиты вскоре после того, как станут доступны. Составьте план для продолжения регулярной установки обновлений по мере их выпуска, чтобы обеспечить свою защиту в будущем.

15

Архитектуры блокчейна и майнинга биткоинов

Эта глава начинается с краткого введения в концепции, связанные с блокчейном, открытым криптографически защищенным реестром, содержащим последовательность транзакций. За введением следует обзор процесса майнинга биткоинов, который добавляет транзакции в блокчейн Bitcoin и вознаграждает тех, кто выполняет эту задачу, оплатой в биткоинах. Для обработки биткоинов требуется высокопроизводительное вычислительное оборудование, которое представлено с точки зрения компьютерной архитектуры для майнинга биткоинов текущего поколения. Глава завершается кратким введением в некоторые альтернативные биткоину криптовалюты.

После прочтения этой главы вы получите представление о концепции блокчейна и о том, как используется эта технология. Вы изучите этапы процесса майнинга биткоинов и ознакомитесь с ключевыми характеристиками компьютерных архитектур для майнинга биткоина и особенностями некоторых популярных в настоящее время криптовалют.

В этой главе будут представлены следующие темы:

- введение в блокчейн и биткоин;
- процесс майнинга биткоинов;
- компьютерные архитектуры для майнинга биткоинов;
- альтернативные виды криптовалют.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Введение в блокчейн и биткоин

Концепция **биткоина** впервые стала достоянием общественности в статье Сатоши Накамото (Satoshi Nakamoto), вышедшей в 2008 г. под названием "Bitcoin: пиринговая электронная денежная система" ("Bitcoin: A Peer-to-Peer Electronic Cash System"). Имя автора, по-видимому, было псевдонимом, и личность автора (или авторов) статьи осталась неизвестной для публики. В статье излагались математическая и криптографическая основы системы для выполнения децентрализованных финансовых транзакций.

Функционирование централизованной финансовой системы зависит от таких организаций, как правительства и банки, которые осуществляют мониторинг и контроль деятельности системы и регулируют то, что разрешено делать пользователям системы.

Концепция биткоина не имеет централизованного регулятора и полностью полагается на равноправные сетевые узлы, конкурентное взаимодействие которых организовано таким образом, чтобы поддерживать стабильную работу системы. Любой желающий может присоединиться к сети в качестве равноправного участника и немедленно получить все привилегии, доступные другим участникам сети.

Одна из важных особенностей сети Bitcoin заключается в том, что отдельным лицам, использующим эту валюту, не обязательно верить в то, что все, кто взаимодействует с экосистемой Bitcoin, будут действовать честно или достойно. Если добропорядочные равноправные узлы контролируют большую часть вычислительной мощности, доступной сети Bitcoin, то пользователи могут быть уверены в надежности системы.

Конечно, эта гарантия надежности основывается на допущении, что единственный реальный путь для злоумышленников взломать реестр биткоинов или извлечь данные о содержащихся в нем транзакциях — это поставить под угрозу целостность системы, основанную на консенсусе. Если программная ошибка в коде системы Bitcoin становится причиной уязвимости, которую могут использовать злоумышленники, или если алгоритм шифрования, на который опирается система, оказывается слабым, то могут существовать и другие способы взломать систему.

Информация о транзакциях системы Bitcoin хранится в **блокчейне** — распределенном реестре, содержащем криптографически защищенные записи обо всех транзакциях, которые проводились с биткоинами с момента появления системы. Каждый равноправный узел сети может в любое время запросить и получить полную копию блокчейна. В ходе процесса ввода в работу только что присоединившегося к сети

равноправного участника новая система должна загрузить и проверить все транзакции блокчейна, начиная с самой первой, и до конца блокчейна и самого последнего добавленного блока. Это обязательный шаг для того, чтобы новый равноправный участник подтвердил все транзакции вплоть до текущего состояния блокчейна.

Для хранения своих средств пользователи сети Bitcoin полагаются на программные приложения, называемые цифровыми кошельками. **Цифровой кошелек** отслеживает баланс биткоинов своего владельца и помогает выполнять переводы другим пользователям сети Bitcoin и получать переводы от них.

В кошельке хранится секретный ключ, используемый владельцем для доступа к принадлежащим ему биткоин-средствам. Если хакер получит доступ к секретному ключу биткоин-кошелька, он сможет перевести хранящиеся в кошельке средства куда пожелает.

Когда пользователь сети Bitcoin инициирует транзакцию для перевода некоторого количества биткоинов другому пользователю или получения биткоинов от него, в блокчейн должна быть добавлена запись об этом, которая помимо одобрения отправителя транзакции должна быть проверена равноправными участниками сети.

Сам блокчейн является общедоступным, но содержащаяся в нем информация не идентифицирует отправителя и получателя, участвующих в конкретной транзакции, каким-либо иным способом, кроме как путем предоставления цифрового ключа, связанного с цифровым кошельком конкретного пользователя. Этот номер не связан с личностью пользователя каким-либо иным образом кроме ситуации, когда тот же номер используется для транзакции с другим лицом, которому известна личность владельца данного кошелька. Для того чтобы избежать такой потери анонимности, пользователь при желании может создавать новый идентификатор кошелька для каждой новой транзакции. Благодаря этой частичной анонимности киберпреступники предпочитают использовать биткоин для таких целей, как получение выкупа, затребованного с помощью программ-вымогателей.

Для того чтобы конвертировать некоторое количество биткоинов в более традиционную валюту, такую как доллары США, или обратно, финансовое учреждение, уполномоченное работать в этой валюте, может выполнить эту транзакцию за определенную плату. В качестве альтернативы пользователь может перевести некоторое количество биткоинов другому пользователю в обмен на взаимно согласованное количество наличных денег.

Каждая биткоин-транзакция влечет за собой небольшую (но необязательную) комиссию, выплачиваемую первому майнеру биткоинов, который успешно выполнит работу, необходимую для добавления набора транзакций от разных пользователей, включая транзакцию данного пользователя, в блокчейн. Сетевые узлы, которые выполняют эти вычисления, называют **системами майнинга биткоинов**, а людей, которые владеют выполняющими эту работу компьютерными системами и управляют ими, называются **майнерами биткоинов**. Добровольное включение комиссии в каждую транзакцию повышает вероятность того, что майнеры включат транзакцию в следующий блок, тем самым ускорив ее прохождение.

Добавление блока в блокчейн — это задача, намеренно требующая больших вычислительных затрат. Узлы майнинга соревнуются за то, чтобы первыми успешно выполнить вычисления, необходимые для добавления блока в блокчейн. Первый узел, продемонстрировавший правильное решение для нового блока, получает комиссию, связанную с блоком, а также все комиссии, предлагаемые за транзакции, содержащиеся в данном блоке. Более подробно мы обсудим майнинг биткоинов позже в этой главе.

Как следует из названия, блокчейн (blockchain) — это цепочка блоков. Каждый блок содержит криптографически защищенные описания множества биткоин-транзакций между пользователями сети.

Начиная с первого блока, размещенного в начале цепочки при ее создании, каждый последующий блок содержит ссылку на блок, непосредственно предшествующий ему в цепочке. Криптографические методы используются для того, чтобы сохранить неизменными все блоки и транзакции внутри них, а также исключить возможность подделки ссылок между блоками.

На рис. 15.1 показано упрощенное представление конца цепочки блокчейна после добавления в нее нового блока с обозначением "Блок *X*". Каждый блок содержит криптографически защищенные ссылки на сведения о каждой содержащейся в нем транзакции.

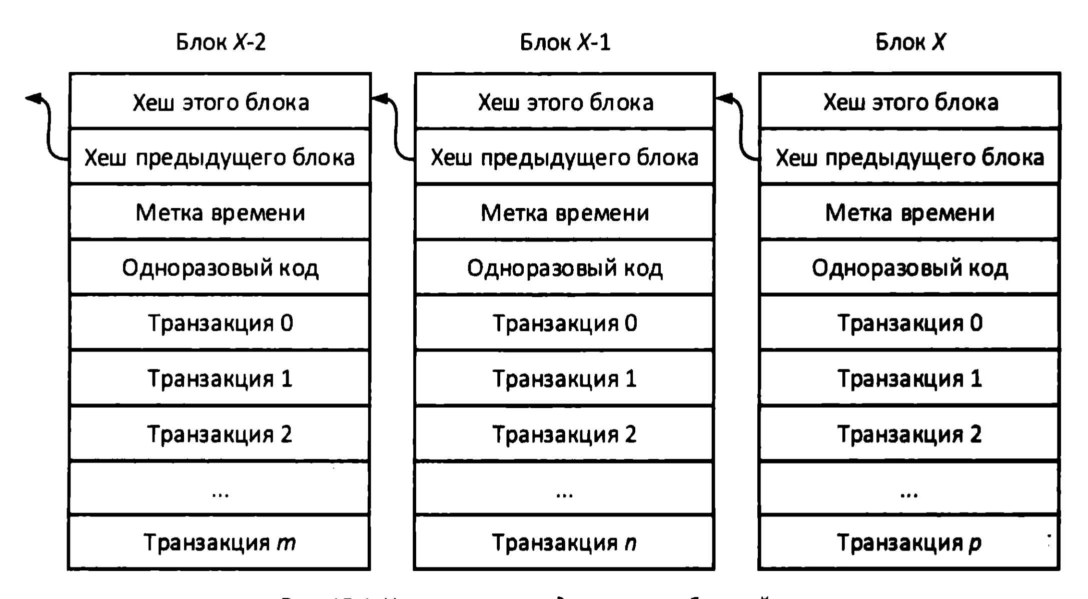


Рис. 15.1. *Упрощенное представление блокчейна*

Каждый блок пронумерован для указания его места в цепочке. На момент написания этого текста в цепочке насчитывается более 700 000 блоков. Новый блок создается примерно каждые 10 минут. Блок содержит транзакции, которые были инициированы в течение предыдущего интервала и собраны для формирования списка транзакций данного блока. Это означает, что для "клиринга" транзакции обычно требуется не менее 10 минут, используя аналогию с размещением чека на банковском счете.

Количество транзакций, включенных в каждый блок, меняется с течением времени в зависимости от объема транзакций, инициированных пользователями.

Для данных каждого блока в сочетании с хешем предыдущего блока в цепочке вычисляется криптографическая хеш-функция. Затем майнеры конкурируют, чтобы определить 32-битное значение (называемое **одноразовым кодом (nonce)**), которое может быть помещено в блок и позволяет получить хеш-значение блока, не превышающее в численном выражении целевого хеш-значения, предоставляемого программным обеспечением сети Bitcoin. Процесс добавления блоков в цепочку включает в себя сначала поиск одноразового кода, который обеспечивает выполнение условий сравнения с целевым хешем сети, затем публикацию нового блока в сети и, наконец, получение подтверждения от нескольких равноправных узлов о том, что новый блок на самом деле действителен.

Целевой хеш сети Bitcoin меняется с течением времени с целью поддержания стабильности обработки транзакций. Это означает, что объем работы, которую майнер должен выполнить, чтобы получить доход, меняется с течением времени и зависит от таких факторов, как объем вычислительной мощности для майнинга, задействованный в настоящее время в сети.

Поскольку сеть одноранговая, добавляемые блоки должны быть проверены равноправными участниками сети с целью подтвердить, что каждый новый блок содержит одноразовый код, при котором хеш блока не превышает значения целевого хеша, и что содержащаяся в блоке информация в остальном верна. Как только между участниками будет достигнут консенсус о том, что блок действителен и что он стал победителем, т. е. первым блоком с действительным одноразовым кодом, этот блок добавляется в блокчейн.

Архитектура блокчейна доказала свою надежность при наличии угроз безопасности. Попытки вставить недействительные блоки в блокчейн легко обнаруживаются с помощью проверки хеша, и любые недействительные блоки отбрасываются.

По замыслу системы в ней может быть создан только 21 млн биткоинов. На сегодняшний день в обращении находится более 18 млн, осталось добыть менее 3 млн. По некоторым оценкам, чтобы добыть оставшиеся биткоины, потребуется более 120 лет.

В программном обеспечении Bitcoin Core предусмотрен ряд мер для поддержания стабильности сети и достижения цели ограничения общего количества биткоинов 21 млн.

- За счет изменения целевого хеша сети алгоритм пытается поддерживать 10-минутный интервал создания блоков. Если бы майнинг каждого блока занимал ровно 10 минут, то за каждые две недели было бы добыто 2016 блоков (6 блоков в час × 24 часа × 14 дней). Поскольку время майнинга для каждого блока меняется в зависимости от доступной в сети вычислительной мощности, ПО Bitcoin Core с интервалом, соответствующим добавлению каждых 2016 блоков, меняет целевой хеш на расчетное значение, при котором на майнинг предыдущих 2016 блоков ушло бы две недели. Затем этот целевой хеш используется для майнинга следующих 2016 блоков.

* 3 января 2009 г. Сатоши Накамото добыл первый блок блокчейна сети Bitcoin. Этот блок, которому был присвоен номер 0 в цепочке, называют *первичным блоком сети Bitcoin*. Вознаграждение майнера за этот блок составило 50 биткоинов. Такое же вознаграждение было предоставлено майнерам первых 210 000 блоков. Затем вознаграждение за блок было уменьшено вдвое, до 25 биткоинов за следующие 210 000 блоков. Вознаграждение уменьшается вдвое после каждого последующего набора из 210 000 блоков. Если бы майнинг каждого блока занимал ровно 10 минут, то на майнинг 210 000 блоков ушло бы четыре года. В 2021 г. вознаграждение за блок составляло 6,25 биткоина, снизившееся вдвое с уровня в 12,5 биткоинов 11 мая 2020 г. Этот процесс деления пополам гарантирует, что общее количество биткоинов будет ограничено 21 млн.



ТЕХНОЛОГИЯ БЛОКЧЕЙНА

Сеть Bitcoin использует технологию блокчейна для ведения криптографически защищенного реестра биткоин-транзакций, но это не единственное применение блокчейна. Блокчейн предоставляет более широкие возможности, которые можно использовать в любой области, где необходимо надежно отслеживать серию транзакций с течением времени. Например, блокчейн можно было бы использовать в библиотеке для записи событий выдачи и возврата книг.

В самые первые дни майнинга биткоинов (май 2010 г.) майнер Ласло Ханеч (Laszlo Hanyecz), как известно, купил две пиццы за 10 000 биткоинов. Похоже, это было первое использование данной криптовалюты для покупки физических товаров. Это событие стало рассматриваться как ключевой момент в развитии платежной системы Bitcoin и использовании ее распределенного реестра на основе блокчейна. На момент написания статьи 10 000 биткоинов могли стоить около 500 млн долларов.

Подсчитано, что 4 млн биткоинов были безвозвратно потеряны их владельцами. Биткоины можно потерять, если владелец потеряет секретный ключ к кошельку, где они хранятся, например путем удаления всех копий этого ключа или посредством уничтожения единственного жесткого диска, содержащего ключ. Потерянные биткоины по-прежнему принадлежат своему владельцу, но никто не может их восстановить и использовать.

Далее мы рассмотрим алгоритм безопасного хеширования, который обеспечивает сети Bitcoin криптографическую защиту и формирует ядро процесса майнинга биткоинов.

Алгоритм хеширования SHA-256

Фундаментальной операцией, лежащей в основе вычислений, используемых при майнинге биткоинов и многих других криптовалют, является безопасное хеширо-

вание. В качестве хеш-алгоритма в системе Bitcoin используется SHA-256 — опубликованный стандартный криптографический алгоритм хеширования, который был законодательно закреплен правительством США в Публикации 180-4 **Федерального стандарта обработки информации** (Federal Information Processing Standards, FIPS).

SHA-256 работает с блоками данных, длина которых кратна 512 битам. Этот алгоритм определяет процедуру добавления битов заполнения к данным для достижения требуемой длины.

Результатом вычисления SHA-256 является 256-битное хеш-значение, которое чаще всего представляется в виде 64 шестнадцатеричных символов. Ниже перечислены наиболее важные особенности взаимосвязи между блоком входных данных и хеш-значением этого блока на выходе SHA-256.

- Длина хеш-значения на выходе всегда равна 256 битам, независимо от размера блока входных данных. Длина входных данных может быть меньше или намного больше 256 бит.
- Вычисление хеш-значения конкретного блока данных с помощью алгоритма SHA-256 всегда дает один и тот же результат.
- Изменение любой части блока данных, даже одного бита, обычно приводит к получению совершенно другого хеша SHA-256 по сравнению с хешем исходного блока данных.
- Теоретически возможно внести изменения в блок данных таким образом, чтобы в результате получить то же хеш-значение SHA-256, что и для неизмененной исходной версии, однако на практике невозможно создать два разных блока данных, которые дают один и тот же результат хеширования с помощью алгоритма SHA-256.

Когда хеширование двух разных блоков данных дает один и тот же результат, возникает **хеш-конфликт**. В сфере криптографических хеш-функций возможность конфликтов представляет собой угрозу безопасности. Эффективный и безопасный криптографический алгоритм хеширования должен гарантировать чрезвычайно низкую вероятность хеш-конфликтов. Для алгоритма SHA-256 мы можем считать вероятность хеш-конфликта астрономически малой величиной.

Причина, по которой так трудно найти два разных блока данных, которые дадут одно и то же хеш-значение SHA-256, заключается в объеме работы, которую необходимо выполнить для определения содержимого второго блока, чтобы его хеш совпадал с хешем первого блока. Прямолинейный подход к определению второго блока данных, который генерирует тот же хеш, что и первый блок, заключается в использовании процедуры полного перебора.

Алгоритм полного перебора для поиска блока данных, соответствующего заданному (целевому) хеш-значению, может обрабатывать второй блок данных как последовательность битов, которые мы интерпретируем как очень длинное целое число длиной 256 бит. Ниже приведен примерный порядок действий для поиска блока данных, который создает заданный хеш с использованием метода полного перебора:

1. Ввести целевое хеш-значение, для которого следует подобрать блок данных.

2. Обнулить все биты в блоке данных.
3. Вычислить хеш-значение блока данных с помощью алгоритма SHA-256.
4. Совпадает ли вычисленный хеш блока с целевым хешем? Если да, выйти и отобразить соответствующий блок данных. Если нет, продолжить выполнение с *шага 5*.
5. Увеличить целое число в блоке данных на единицу.
6. Перейти к *шагу 3*.

Описанный здесь алгоритм поиска хеш-конфликта методом перебора прост и в конечном счете найдет соответствующее хеш-значение (если оно существует), однако совершенно нереалистично ожидать, что он когда-либо вернет полезный результат для хеша SHA-256 достаточно большого блока данных (длиной 256 бит или более).

Наш алгоритм поиска методом перебора должен выполнить цикл (шаги 3–6) 2^{256} раз, чтобы иметь почти достоверный шанс найти хеш-конфликт. Сколько времени это займет? Ответ включает в себя всю вычислительную мощность, доступную на Земле (сейчас и в будущем), и все время, оставшееся до того момента, когда погаснет Солнце. При этом, весьма вероятно, что за это время не удастся сколько-нибудь заметно приблизиться к блоку данных, который даст нужное хеш-значение.

Другими словами, учитывая текущее состояние вычислительных возможностей и разумные прогнозы относительно будущего роста этих возможностей, весьма вероятно, что в обозримой перспективе алгоритм SHA-256 не будет подвержен риску уязвимости из-за хеш-конфликтов, если, конечно, кто-то не обнаружит в нем другую уязвимость. А в области криптографических алгоритмов такой риск существует всегда.

Далее мы рассмотрим порядок действий для вычисления хеша SHA-256 блока данных.

Вычисление хеша SHA-256

Алгоритм SHA-256 работает с блоком данных, который может иметь любую длину от 1 до $2^{64} - 1$ бит. Блок данных, также называемый сообщением, в этом обсуждении рассматривается как линейная строка битов.

Прежде всего сообщение дополняется до длины, кратной 512 битам, с использованием следующей процедуры. Дополнение сообщения выполняется, даже если длина исходного сообщения кратна 512 битам.

1. Добавить 1 бит к сообщению.
2. Добавить к сообщению минимальное количество нулевых битов таким образом, чтобы длина сообщения в битах была на 64 меньше, чем значение, кратное 512.
3. Добавить к сообщению 64-битное целое число без знака, содержащее длину исходного сообщения в битах.

Структурными элементами алгоритма SHA-256 являются простые логические и математические операции: И, ИЛИ, исключающее ИЛИ, НЕ, сложение целых чисел, сдвиг вправо, вращение вправо и объединение битов. Эти операции выполня-

ются над 32-битными словами. При выполнении сложения флаг переноса процессора игнорируется.

Алгоритм SHA-256 определяет несколько более сложных операций, которые смешивают простые структурные элементы, упомянутые в предыдущем абзаце, с предопределенными константами для кодирования данных входного блока и получения случайного на вид 256-битного значения в качестве выходного хеш-значения.

Алгоритм SHA-256 обрабатывает каждую 512-битную секцию входных данных последовательно. После дополнения сообщения 512 бит каждой секции разделяются на 64 слова по 32 бита в каждом. Обработка каждой секции, по сути, представляет собой процедуру кодирования, которая многократно смешивает биты в словах посредством серии логических операций. Алгоритм выполняет цикл из 64 проходов, содержащий серию интенсивных логических и математических операций над словами данных в каждой секции.

Подводя итог, можно сказать, что вычисление хеша SHA-256 даже для небольшого блока данных требует весьма значительной последовательности вычислений. В алгоритме принципиально не существует коротких путей, которые позволили бы пропустить какой-либо из этапов вычислений.

Пример входных и выходных данных SHA-256: входной блок данных, состоящий только из символов ASCII abc, после дополнения и выполнения вычислительных операций SHA-256 даст на выходе следующие 64 шестнадцатеричные цифры в качестве хеш-значения:

```
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

В следующем разделе мы обсудим некоторые ключевые атрибуты исходного кода, на основе которого работает сеть Bitcoin. Этот код носит название Bitcoin Core.

Программное обеспечение Bitcoin Core

Любой, у кого есть компьютер и доступ в Интернет, может настроить у себя узел сети Bitcoin. Для того чтобы получить полный доступ к возможностям, предлагаемым сетью Bitcoin, владельцу компьютера необходимо настроить полный узел. **Полный узел** Bitcoin выполняет операции, необходимые для проверки транзакций и блоков. Этот процесс включает в себя прием транзакций и блоков от других полных узлов, проверку каждой транзакции и блока и пересылку блоков и результатов проверки другим полным узлам.

Bitcoin Core имеет открытый исходный код, который можно загрузить по адресу **https://bitcoincore.org/en/download/**. Этот код работает на компьютерных системах с ОС Windows, macOS X и Linux, которые удовлетворяют стандартным требованиям к памяти и дисковому пространству и имеют широкополосное подключение к Интернету. Исполняемые приложения Bitcoin Core можно загрузить и установить напрямую или сгенерировать их на основе исходного кода.

После установки и запуска приложение Bitcoin Core загрузит весь блокчейн Bitcoin с равноправных узлов, начиная с первичного блока и заканчивая самыми послед-

ними добавленными блоками. По состоянию на 2021 г. полный объем данных блокчейна составляет более 400 Гбайт. Приложение Bitcoin Core может удалить ранние блоки после того, как они будут проанализированы, чтобы освободить часть дискового пространства. Первоначальная загрузка и анализ ранних блоков необходимы для проверки всей истории блокчейна и подтверждения действительности и точности текущего состояния последних блоков.

Если вы решите ограничить использование диска программным обеспечением Bitcoin, например размером в 2 Гбайт, то при попытке проверки транзакций, которые выводят средства из транзакций, расположенных в более ранних блоках, вашему узлу придется запрашивать копии более ранних блоков у других равноправных узлов. Каждый раз, когда программа получает данные от (недоверенного) равноправного узла сети, она выполняет проверку хеша и цифровой подписи с целью убедиться, что все элементы в передаваемых данных действительны и заслуживают доверия.

Полный узел также может выступать в качестве клиентского узла. **Клиентский узел** позволяет пользователю системы Bitcoin инициировать биткоин-транзакции с другими пользователями и отвечать на транзакции, инициированные другими пользователями. Большинство пользователей Bitcoin используют клиентский узел, который может представлять собой приложение, запущенное на смартфоне.

Далее мы более подробно рассмотрим требования к обработке при майнинге биткоинов и узнаем, как цена на электроэнергию влияет на получаемую майнером прибыль.

Процесс майнинга биткоинов

Вычислительная сложность алгоритма SHA-256 напрямую связана с возможностью извлечения прибыли из майнинга биткоинов. Единственный способ определить хеш-значение SHA-256 для конкретного блока данных — это выполнить все этапы алгоритма SHA-256 над всеми битами в блоке.

Ключевая особенность процесса майнинга биткоинов заключается в том, что процесс нахождения действительного одноразового кода, который позволит получить хеш-значение блока, не превышающее текущего целевого хеш-значения сети, намеренно сделан очень сложным. В действительности, скорее всего потребуется огромное количество предположений о различных значениях одноразового кода, прежде чем будет найден код, удовлетворяющий цели. Поскольку какая-либо предсказуемая взаимосвязь между содержимым блока и хешем SHA-256 для этого блока отсутствует, нет более эффективного метода определить подходящее значение одноразового кода, чем простое многократное вычисление хеша для блока данных с помощью разных значений одноразового кода, пока не появится хеш, удовлетворяющий целевым критериям.

Процесс поиска одноразового кода, удовлетворяющего требованию по целевому хешу, называют **доказательством работы**. Для того чтобы получить доказательство работы, необходимое для майнинга биткоинов, майнер должен использовать

подходящее оборудование и электроэнергию, а также потратить определенное время для выполнения алгоритма.

Для современного ПК несложно перепробовать все возможные значения 32-разрядного одноразового кода в течение нескольких секунд. К сожалению для майнеров, это вычисление редко позволяет получить хеш-значение, удовлетворяющее целевому значению хеша сети. После исчерпания всех возможных 32-разрядных значений одноразового кода майнер должен изменить пробный блок, с которым он работает. Затем он сможет начать перебор всех возможных значений одноразового кода для содержимого нового блока. Существует несколько приемлемых для сети Bitcoin способов, с помощью которых майнеры могут изменить содержимое пробного блока.

- **Изменение метки времени внутри блока.** Каждый блок в цепочке содержит метку времени, которая отражает время его создания, но сама метка не используется для каких-либо важных целей, таких как определение порядка, в котором были созданы блоки. Если майнер безуспешно исчерпывает все возможные значения одноразового кода, можно изменить метку времени в заголовке блока и снова попробовать все возможные значения одноразового кода. Изменение метки времени, по сути, увеличивает размер пространства для поиска одноразового кода. Возможны лишь небольшие изменения метки времени, но значительные изменения недопустимы.
- **Добавление в блок новых транзакций.** Вполне вероятно, что во время предыдущего неудачного раунда поиска удовлетворительного одноразового кода из сети Bitcoin продолжали поступать транзакции. Запросив новый пробный блок, содержащий недавно добавленные транзакции, майнер также увеличивает размер пространства для поиска одноразового кода.
- **Изменение данных в заголовке транзакции.** Первая транзакция в списке транзакций блока является особенной в том смысле, что она представляет собой платеж майнеру за добавление блока в блокчейн. Майнер может добавить дополнительные данные к этой транзакции. Эти добавленные данные, при их использовании с целью увеличения пространства поиска одноразового кода, называют extraNonce. Обычная процедура работы с extraNonce заключается в проверке хешей блоков для всех возможных значений одноразового кода, после чего, рассматривая extraNonce как целое число, следует увеличить extraNonce на единицу и повторить перебор всех возможных значений одноразового кода.

Процедуры пошагового увеличения одноразового кода и выполнения других манипуляций для увеличения эффективного пространства поиска подходящего одноразового кода, как правило, не занимают значительного времени в процессе майнинга биткоинов. Основная часть работы, связанной с процессом поиска, происходит во время многократного выполнения хеш-алгоритма SHA-256 при проверках различных значений одноразового кода с целью найти хеш блока, который не превышает значения целевого хеша сети.

На заре майнинга биткоинов (примерно в 2010 г.) майнер мог использовать персональный компьютер с достаточно хорошей производительностью, чтобы зарабатывать на майнинге несколько долларов в день. В то время уровень сложности, определяемый целевым хешем сети, был достаточно низким, чтобы аппаратные средства стандартного ПК могли находить значения одноразового кода, удовлетворяющие цели хеша, с разумной вероятностью успеха.

По мере присоединения к сети все большего количества майнеров и повышения производительности их вычислительного оборудования общий объем вычислительной мощности, используемой для получения доказательств работы по добавлению каждого блока в цепочку, продолжал расти.

Алгоритм корректировки целевого хеша сети Bitcoin изменяет его значение после каждых 2016 блоков, чтобы поддерживать добавление одного блока в среднем каждые 10 минут. Это означает, что по мере роста общей вычислительной мощности сети Bitcoin шансы любого майнера-одиночки, даже использующего очень мощный графический процессор, первым найти решение для любого блока значительно уменьшаются. Майнер, скорее всего, вообще не получит никакой отдачи, даже после многолетнего майнинга.

Для того чтобы отдельные лица могли продолжать участвовать в майнинге биткоинов с получением хотя бы некоторой отдачи от вложенных усилий, возникла концепция пулов майнинга биткоинов. Это тема следующего раздела.

Пулы майнинга биткоинов

Пул майнинга биткоинов — это группа майнеров, которые объединяют свои вычислительные мощности, чтобы увеличить шансы на вознаграждение после успешного получения доказательства работы для блоков, добавленных в блокчейн Bitcoin. Присоединяясь к майнинговому пулу, майнер соглашается вкладывать свои вычислительные мощности в пул и получать долю вознаграждений от майнинга, выполняемого участниками пула.

По сути, участники майнингового пула разделяют большую задачу получения доказательства работы на набор более мелких задач и передают эти задачи отдельным участникам пула для выполнения. Если один из участников пула правильно решает задачу получения доказательства работы, пул добавляет блок в цепочку и делит вознаграждение между участниками.

Организаторы майнингового пула должны создать центр обработки данных для управления взаимодействием пула с его участниками-майнерами и с сетью Bitcoin. Для этого требуется вычислительное оборудование и персонал для его настройки и управления системой на повседневной основе. За эту услугу операторы майнинговых пулов взимают плату в размере обычно 1–3% прибыли от майнинга.

Присоединение к пулу майнинга биткоинов позволяет майнеру получать некоторую (обычно небольшую) регулярную прибыль в обмен на предоставление вычислительной мощности пулу. Чем больше вычислительной мощности (с точки зрения количества хешей SHA-256, вычисляемых за единицу времени) вносит майнер, тем больше будет его вознаграждение, когда данный пул успешно добавит блок в цепочку.

Вероятность успеха конкретного пула майнинга биткоинов в течение определенного периода можно количественно оценить по скорости вычисления хешей для данного пула относительно аналогичной скорости для всей сети Bitcoin. **Скорость вычисления хешей** — это количество хешей SHA-256, рассчитываемых в секунду, где каждая операция хеширования является попыткой получить доказательство работы для блока.

В течение 2021 г. общая скорость вычисления хешей в сети Bitcoin по разным оценкам колебалась между 80 и 180 млн терахешей в секунду (Тх/с). Один **терахеш** — это один триллион хешей или 10^{12} хешей. Для обозначения одного миллиона терахешей используется термин **"эксахеш"** (Эх), равный 10^{18} хешей. Эта единица несколько упрощает обсуждение скорости расчета хешей в 2021 г., которая менялась в промежутке от 80 до 180 эксахешей в секунду. Этот диапазон также можно выразить как 80–180 Эх/с.

Доля общей скорости вычисления хешей сети, контролируемая пулом майнинга, определяет ожидаемую частоту добавления блоков для данного пула. Как мы видели, майнинг нового блока занимает в среднем 10 минут. Следующее уравнение показывает, как часто майнинговый пул может рассчитывать на успех при майнинге блока, исходя из его доли в общей скорости хеширования сети:

$$T_B = \frac{10\text{ минут}}{\dfrac{H_P}{H_N}}. \tag{15.1}$$

В уравнении (15.1) T_B представляет среднее время между блоками, добытыми пулом (в минутах), H_P — скорость вычисления хешей для пула, H_N — общую скорость вычисления хешей для сети.

Это уравнение справедливо, т. к. процесс получения доказательства работы над блоком является статистическим по своей природе, где каждый обрабатывающий элемент сети выдвигает серию предположений, и каждое предположение, сделанное каждым участником, имеет одинаковые, очень маленькие, но равные шансы на успех.

Если пул контролирует 0,1% (или одну тысячную) общей скорости вычисления хешей сети, то согласно этой формуле он будет добывать новый блок в среднем каждые 10 000 минут, т. е. примерно раз в неделю. При текущей цене биткоина в 45 000 долларов и вознаграждении за блок в размере 6,25 биткоина итоговое вознаграждение за блок составит 281 250 долларов плюс любые комиссии за транзакции, предлагаемые пользователями сети Bitcoin.

КОМИССИЯ ЗА ТРАНЗАКЦИИ



Комиссия за транзакции — это добровольные платежи, предлагаемые пользователями сети Bitcoin в качестве стимула для повышения уровня приоритета своих транзакций при их размещении в блоках во время майнинга. Каждый раз, когда пользователь инициирует новую биткоин-транзакцию, у него есть возможность выделить часть задействованных в транзакции средств в счет комиссии за эту транзакцию.

Майнеры могут выбирать, какие транзакции они включают в каждый блок, над которым работают. Это означает, что они обычно предпочитают транзакции, за которые предлагается более высокая комиссия. В периоды высокой частоты транзакций предложение низкой комиссии за транзакцию может привести к более длительному ожиданию ее добавления в блокчейн.

В конце концов, когда количество монет, оставшихся для добычи, приблизится к нулю, плата за транзакции станет единственным стимулом для майнеров продолжать обрабатывать транзакции и добавлять блоки в цепочку. В течение 2021 г. еженедельная медианная (медиана — число из набора, половина значений в котором меньше, а другая половина значений — больше этого числа) комиссия за биткоин-транзакцию менялась от 0,27 до 26,96 доллара.

После успешного добавления нового блока в цепочку менеджер майнингового пула должен разделить вознаграждение за этот блок между участниками пула. Для того чтобы отслеживать долю доказательства работы, приходящуюся на каждого майнера в пуле, программа управления пулом устанавливает для своих майнеров целевое значение хеша, которое значительно выше (условие, которое легче выполнить) целевого хеша сети. Это означает, что участники пула будут возвращать менеджеру пула множество хеш-решений, которые удовлетворяют целевому уровню пула, но не удовлетворяют целевому уровню сети. Отслеживая, сколько хешей, удовлетворяющих цели пула, возвращает каждый майнер, менеджер пула может определить, сколько хешей было вычислено каждым участником пула. После этого вознаграждение за блок распределяется между участниками пула пропорционально их вкладу в общую работу по расчету хеш-значений.

При типичной для 2021 г. скорости вычисления хешей сети в 140 Эх/с пул должен работать со скоростью 140 000 терахешей в секунду, или 140 000 Тх/с, чтобы контролировать 0,1% скорости вычисления хешей сети. Это определенно выглядит как достаточно большое количество хешей. Для того чтобы получить некоторое представление о вычислительной мощности, необходимой для достижения такой скорости вычисления хешей, давайте для начала обсудим использование процессоров стандартных ПК для выполнения операции хеширования. Мы рассмотрим этот вопрос с точки зрения одиночного майнера, который не участвует в майнинговом пуле.

Майнинг с помощью центрального процессора

Если вы не хотите платить комиссию, требуемую от участников пула майнинга биткоинов, вы можете заняться майнингом самостоятельно, используя одно или несколько вычислительных устройств, находящихся в вашей собственности или под вашим контролем. Такой подход называют **соло-майнингом**. Мы можем оценить доходы от соло-майнинга на основании возможностей хеширования оборудования, выделенного для этой цели.

Одним из самых производительных процессоров, доступных в настоящее время, является AMD Ryzen Threadripper 3970X. Он имеет 32 ядра и поддерживает 64 одновременных потока с тактовой частотой, которая колеблется от 3,7 до 4,5 ГГц. Большое количество одновременных потоков позволяет параллельно вычислять хеши для нескольких значений одноразового кода. Если судить по сравнительным тестам, процессор 3970X может вычислять около 19 900 биткоин-хешей в секунду.

Рассмотрим ПК, содержащий один процессор AMD Ryzen Threadripper 3970X. Мы можем подставить скорость вычисления хешей этого процессора в уравнение (15.1) и оценить интервал между успешной добычей блоков. Предположим, что номинальная сетевая скорость вычисления хешей для 2021 г. составляет 140 Эх/с. Результат этого вычисления показан в уравнении (15.2).

$$T_B = \frac{10 \text{ минут}}{\left[\dfrac{19{,}8 \times 10^3}{140 \times 10^{18}}\right]} = 7{,}04 \times 10^{16} \text{ минут.} \tag{15.2}$$

Из этого уравнения мы видим, что средний интервал между успешно добытыми блоками на одном 3970X равен $7{,}04 \times 10^{16}$ минут, что составляет около 133 млрд лет. Очевидно, что эта конфигурация не является жизнеспособной для тех, кто пытается получить хоть какую-то отдачу от майнинга биткоинов.

Вскоре после первого представления концепции биткоина была выпущена версия программы для майнинга с открытым исходным кодом, которая использовала возможности параллельной обработки аппаратных средств графических процессоров (GPU). Мы рассмотрим этот вопрос в следующем разделе.

Майнинг с помощью графического процессора

Основным вычислением для майнинга биткоинов является хеш-алгоритм SHA-256.

Задача проверки очень большого количества значений одноразового кода идеально подходит для архитектуры компьютера с параллельной обработкой, поскольку каждая такая проверка не зависит от всех остальных. Из-за этого присущего данному процессу параллелизма переход в контекст GPU был естественным шагом для программного обеспечения майнинга биткоинов. При запуске на графическом процессоре код майнинга в полной мере использует большое количество элементов обработки для выполнения алгоритма хеширования с гораздо более высокой скоростью, чем может достичь центральный процессор даже с большим количеством ядер.

Высокопроизводительный графический процессор способен достичь гораздо более высокой скорости при работе с алгоритмом SHA-256, чем микропроцессор. В первые годы существования биткоина многие майнеры использовали высокопроизводительные графические процессоры для майнинга биткоинов. Примерно до 2014 г. майнеры могли с выгодой использовать растущую вычислительную мощность графических процессоров для выполнения вычислений со скоростью около 1 гигахеша в секунду, или 1 Гх/с, что равно 10^9 хешей в секунду.

Если мы подставим это значение скорости в уравнение (15.1), то сможем найти средний интервал успешного добавления блоков для 2021 г.:

$$T_B = \frac{10\text{ минут}}{\left[\dfrac{1\times10^9}{140\times10^{18}}\right]} = 1{,}40\times10^{12}\text{ минут}. \qquad (15.3)$$

Результатом этого вычисления является среднее значение $1{,}40 \times 10^{12}$ минут, что составляет более 2,6 млн лет. Этот временнóй интервал успешного добавления блоков намного лучше, чем при использовании лишь одного мощного центрального процессора, но это все равно совсем не то, что любой вменяемый майнер-одиночка попытался бы сделать в 2021 г.

Использование графических процессоров для майнинга биткоинов стало невыгодным, когда на рынок были выведены устройства на основе специализированных интегральных схем (ASIC), единственной целью которых являлся майнинг с гораздо более высокой скоростью вычисления хешей, чем могли предложить процессоры общего назначения или графические процессоры. Вычислительные системы, в которых используются специализированные интегральные схемы, являются предметом следующего раздела.

Компьютерные архитектуры для майнинга биткоинов

Алгоритм SHA-256 представляет собой четко определенную последовательность многократно повторяющихся простых логических операций с 32-разрядными элементами данных. Этот алгоритм не требует большого объема памяти при работе с блоками небольшого размера (заголовки биткоин-блоков всегда имеют длину 80 байт). Задачи такого типа являются идеальными кандидатами для оптимизации производительности с использованием цифровых аппаратных средств, специально предназначенных для этой области применения.

Входными данными для этого алгоритма майнинга является заголовок блока-кандидата, содержащий перечисленные ниже элементы данных.

- **Номер версии Bitcoin (4 байта)** — это поле определяет версию программного обеспечения Bitcoin Core. Майнер выбирает номер версии, совместимой с программным обеспечением для майнинга биткоинов.
- **Хеш предыдущего блока (32 байта)** — хеш-значение предыдущего блока в блокчейне. Это значение извлекается из сети Bitcoin как хеш текущего последнего блока в блокчейне.
- **Корень дерева Меркла (32 байта)** — это хеш-значение защищает все транзакции в блоке-кандидате. Термин "корень дерева Меркла" относится к древовидной структуре данных, которая начинается с хешей отдельных биткоин-транзакций и объединяет эти хеши таким образом, чтобы обеспечить эффек-

тивную и безопасную проверку целостности каждой отдельной транзакции в дереве.

- **Время (4 байта)** — метка времени блока в секундах, прошедших с 1 января 1970 г. по **всемирному координированному времени** (Coordinated Universal Time, UTC). Допустимая метка времени блока должна находиться в пределах трехчасового окна относительно текущего времени, определяемого сетью Bitcoin. Это окно допустимости позволяет некоторую корректировку времени блока для увеличения пространства поиска хеша. Ввиду такой гибкости нельзя считать, что метки времени блоков в блокчейне отражают точное время создания блока.
- **Биты (4 байта)** — это поле определяет сложность целевого хеша сети. Здесь размещается значение с плавающей запятой, представленное 24-битной мантиссой и 8-битным порядком в уникальном формате, принятом в сети Bitcoin. Это значение предоставляется сетью Bitcoin.
- **Однократный код (4 байта)** — поле, значение которого майнер изменяет, пытаясь сгенерировать различные хеши.

Эти поля объединяются, формируя 80-байтовый заголовок блока-кандидата. После установки допустимых значений для всех шести параметров майнер вычисляет хеш заголовка и сравнивает его с целевым хешем сети. Если вычисленное значение хеша равно или меньше целевого значения хеша сети, определенного в поле `bits` (биты) заголовка, блок считается действительным, и майнер может отправить его в сеть для проверки и добавления в блокчейн.

ДВОЙНОЙ АЛГОРИТМ SHA-256



Алгоритм хеширования биткоин-блока фактически дважды выполняет SHA-256 для вычисления хеша заголовка блока. Сначала он вычисляет хеш заголовка 80-байтового блока, а затем вычисляет хеш для хеша, вычисленного на первом шаге.

Это вычисление может быть выражено в виде SHA-256(SHA-256(заголовок)).

Почти любое предположение о значении одноразового кода дает хеш, не отвечающий условию для действительного блока. Майнер систематически изменяет одноразовый код и, возможно, метку времени (в допустимых пределах), а также другие части блока, пытаясь вычислить хеш, который обеспечит получение действительного блока. Бóльшая часть работы в этом процессе приходится на повторение операций алгоритма SHA-256.

Первым шагом в разработке специализированных аппаратных средств для майнинга биткоинов является использование в этих целях **программируемой логической интегральной схемы** (ПЛИС — field-programmable gate array, FPGA). Это тема следующего раздела.

Майнинг с помощью ПЛИС

ПЛИС, описанные в *главе 2*, предоставляют средства для создания аппаратной схемы, оптимизированной для решения конкретной задачи, путем подключения к схеме набора универсальных цифровых компонентов, таких как логические вентили, триггеры и регистры. Используя язык описания аппаратных средств (HDL), разработчик может определить логическую последовательность выполнения алгоритма SHA-256, а компилятор HDL превратит эту спецификацию в схему, которую можно загрузить в ПЛИС.

Для того чтобы это решение было полезным для майнинга биткоинов, необходимо включить в его архитектуру некоторую дополнительную логику для управления вводом данных, подлежащих хешированию, и последующего извлечения выходных данных из алгоритма хеширования.

В простейшей реализации этого решения программное обеспечение для майнинга может предоставлять алгоритму ПЛИС входные данные, состоящие из заголовка блока-кандидата и текущего пробного значения одноразового кода. После получения входных данных ПЛИС выполняет алгоритм хеширования SHA-256 и возвращает полученное хеш-значение в программное обеспечение майнинга в качестве выходных данных.

На рис. 15.2 показана возможная реализация этого простого подхода к майнингу.

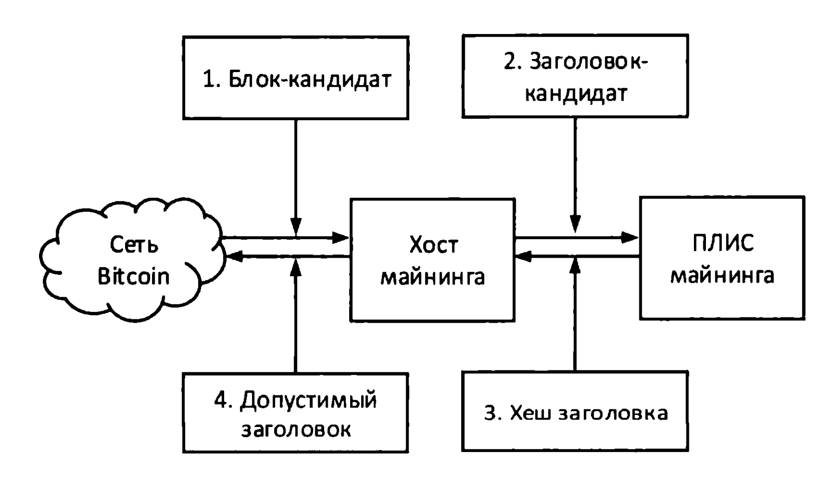


Рис. 15.2. *Простая реализация майнинга на основе ПЛИС*

Последовательность операций для такой конфигурации майнинга состоит из следующих этапов:

1. Хост майнинга (это может быть стандартный ПК или серверная система) запрашивает и получает из сети Bitcoin блок-кандидат для работы.
2. Хост майнинга передает заголовок блока-кандидата (включая текущее пробное значение одноразового кода) в ПЛИС майнинга для хеширования.

3. ПЛИС выполняет операцию хеширования и возвращает хеш-значение хосту майнинга.
4. Хост майнинга оценивает хеш. Если он удовлетворяет требованиям сети, хост пересылает блок в сеть Bitcoin для включения в блокчейн.

Этот подход предусматривает выполнение всех необходимых для майнинга биткоинов этапов с использованием высокоскоростной ПЛИС, однако описанная здесь конфигурация вряд ли обеспечит достаточную скорость хеширования, способную заинтересовать любого серьезного майнера. Это связано с тем, что затраты ресурсов, необходимые для передачи данных в ПЛИС и из нее для каждой оценки хеша, скорее всего, резко снизят среднюю скорость хеширования.

В улучшенной конфигурации с использованием ПЛИС можно было бы предусмотреть циклическое выполнение операции хеширования над целым диапазоном значений одноразового кода и прерывать этот цикл только при безуспешном завершении проверки для всего диапазона или нахождении удовлетворительного хеш-значения. Диапазон проверяемых значений одноразового кода может быть закодирован во встроенном программном обеспечении ПЛИС, и этот диапазон можно расширить до всех 2^{32} возможных значений одноразового кода. Такая конфигурация похожа на конфигурацию, показанную на рис. 15.2, и при ее использовании ПЛИС могла бы в перспективе достичь скорости хеширования, очень близкой к максимально достижимой.

С использованием подхода, подобного описанному здесь, были разработаны различные варианты ПЛИС для вычисления хешей при майнинге биткоинов. Максимальная скорость хеширования сильно зависит от возможностей чипа ПЛИС, используемого для выполнения алгоритма. С помощью реализаций SHA-256 на основе ПЛИС была достигнута скорость хеширования до нескольких сотен (Мх/с). ПЛИС, вычисляющая хеши со скоростью 500 Мх/с, может протестировать все 2^{32} значения одноразового кода менее чем за 10 секунд.

В 2014 г. скорость хеширования на одном чипе ПЛИС в несколько сотен миллионов хешей в секунду не казалась слишком впечатляющей по сравнению с графическими процессорами, которые могли вычислять хеши со скоростью до 1 Гх/с. Однако существуют и другие факторы, влияющие на экономику майнинга биткоинов. Стандартная ПЛИС может стоить несколько долларов по сравнению с ценой высокопроизводительного графического процессора в несколько сотен долларов. Кроме того, ПЛИС обычно потребляет лишь малую часть от мощности, потребляемой графическим процессором. Как мы увидим позже, энергопотребление системы майнинга является ключевым фактором, помогающим определить, зарабатывает майнер биткоинов деньги или теряет их.

Идея создания компьютера с несколькими ПЛИС для майнинга биткоинов может показаться выигрышным подходом по сравнению с майнингом с помощью центрального или графического процессора, однако использование специализированных интегральных схем (ASIC) вместо ПЛИС дает еще более высокую производительность и может обойтись дешевле, если эти устройства можно продать в достаточном количестве. Майнинг с помощью ASIC — это тема следующего раздела.

Майнинг с помощью ASIC

Специализированная интегральная схема (application-specific integrated circuit, ASIC) — это специально разработанный чип, который реализует определенную функцию или набор функций. Ниже перечислены основные отличительные особенности ASIC по сравнению с процессорами общего назначения, графическими процессорами и ПЛИС.

- ASIC содержит только те схемы, которые необходимы для выполнения ее предполагаемой функции. На кристалле нет никаких дополнительных схем, занимающих место и потребляющих электроэнергию. Благодаря этому отдельный кристалл ASIC меньше по размеру и потребляет меньше энергии, чем ПЛИС общего назначения. Это делает ASIC менее дорогостоящими в производстве и менее затратными в эксплуатации, если они изготавливаются в достаточно большом количестве.
- Изготовление первой копии ASIC обходится очень дорого. Для создания схемы и изготовления первой партии кристаллов требуется огромный объем работы по проектированию самой схемы и производственной линии. Если после начала производства в конструкции обнаруживается серьезная ошибка, пересмотр конструкции для устранения проблемы также обходится очень дорого.
- После того как производственная линия настроена и проверки показали, что изготовленные схемы работают должным образом, производство чипа в больших объемах становится очень недорогим с точки зрения затрат на чип.

По мере роста интереса со стороны майнеров и инвесторов в годы, последовавшие за появлением биткоина, экономика оборудования для майнинга росла, обеспечивая такой уровень спроса, который оправдывал затраты и усилия на производство ASIC для майнинга.

Компания Bitmain (**https://www.bitmain.com/**) является крупнейшим изготовителем оборудования для майнинга биткоинов, которое включает в себя чипы ASIC для майнинга, а также комплектные компьютерные системы для майнинга на основе этих чипов. Помимо разработки оборудования для майнинга Bitmain занимается майнингом в собственных интересах и управляет как минимум двумя майнинговыми пулами: BTC.com (**https://pool.btc.com/**) и AntPool (**https://v3.antpool.com/home**).

Компания Bitmain разработала несколько ASIC для майнинга биткоинов, начиная с чипа BM1380, который был выпущен в ноябре 2011 г. Этот чип может выполнять вычисления со скоростью до 2,8 Гх/с при максимальном рабочем напряжении 1,10 В.

Bitmain также выпустила майнинговый компьютер Antminer S1, который содержал 64 чипа BM1380. Эта система могла работать со скоростью 180 Гх/с при потреблении 360 Вт электроэнергии.

Такая скорость хеширования намного превышает ту, которой можно было бы достичь даже с помощью нескольких графических процессоров или 64 чипов ПЛИС.

Давайте оценим, сколько времени в среднем проходит между успешными добавлениями блоков при использовании Antminer S1.

$$T_B = \frac{10\ \text{минут}}{\left[\dfrac{180\times 10^9}{140\times 10^{18}}\right]} = 7,78\times 10^9\ \text{минут}. \qquad (15.4)$$

Среднее время между успешными добавлениями блоков для Antminer S1 составляет $7,78 \times 10^9$ минут, т. е. около 14 800 лет. Это намного лучше, чем для быстрого графического процессора, но все же не является приемлемой конфигурацией для майнера-одиночки, работающего в 2021 г.

Следом за BM1380 компания Bitmain выпустила серию модернизированных ASIC для майнинга: BM1382 (апрель 2014 г.), BM1384 (сентябрь 2014 г.), BM1385 (август 2015 г.) и BM1387 (май 2017 г.). Для версий ASIC, которые последовали за указанными выше, Bitmain опубликовала меньше подробностей. Общая тенденция в каждой новой итерации ASIC заключалась в увеличении скорости хеширования и снижении энергопотребления в расчете на вычисленное хеш-значение.

В 2021 г. одним из самых быстрых доступных майнеров Bitmain стала система Antminer S19 Pro со скоростью хеширования 110 Тх/с. Она потребляет 3250 Вт и продается примерно за 15 000 долларов. Эта система вмещает три платы хеширования, каждая из которых содержит 114 чипов BM1398.

Все 114 чипов на каждой плате соединены последовательно — такую конфигурацию называют последовательной. S19 Pro имеет плату управления, содержащую процессор и встроенное ПО, которое взаимодействует с каждой из хеш-плат и хеш-чипов на ней и управляет их работой. Протокол для связи с чипами представляет собой последовательный формат данных, аналогичный тому, который используется стандартными последовательными портами на компьютерах и других цифровых устройствах.

Плата управления отправляет команды через последовательный интерфейс на первый чип ASIC BM1398 в последовательной цепочке, тот передает ту же команду на следующий чип BM1398 в цепочке и т. д.

Каждый чип BM1398 имеет набор аппаратных адресных строк, которые уникальным образом идентифицируют его в последовательной цепочке. Чип ASIC использует эти адресные строки для определения выделенной ему части пространства поиска одноразового кода.

Каждый чип BM1398 может размещать на шине сообщения, которые проходят по последовательной цепи и принимаются платой управления. Основным типом сообщения, генерируемого чипом хеширования, является уведомление о том, что данный чип получил одноразовый код, который удовлетворяет целевому требованию хеширования.

На рис. 15.3 показана общая конфигурация компонентов в Antminer S19 Pro.

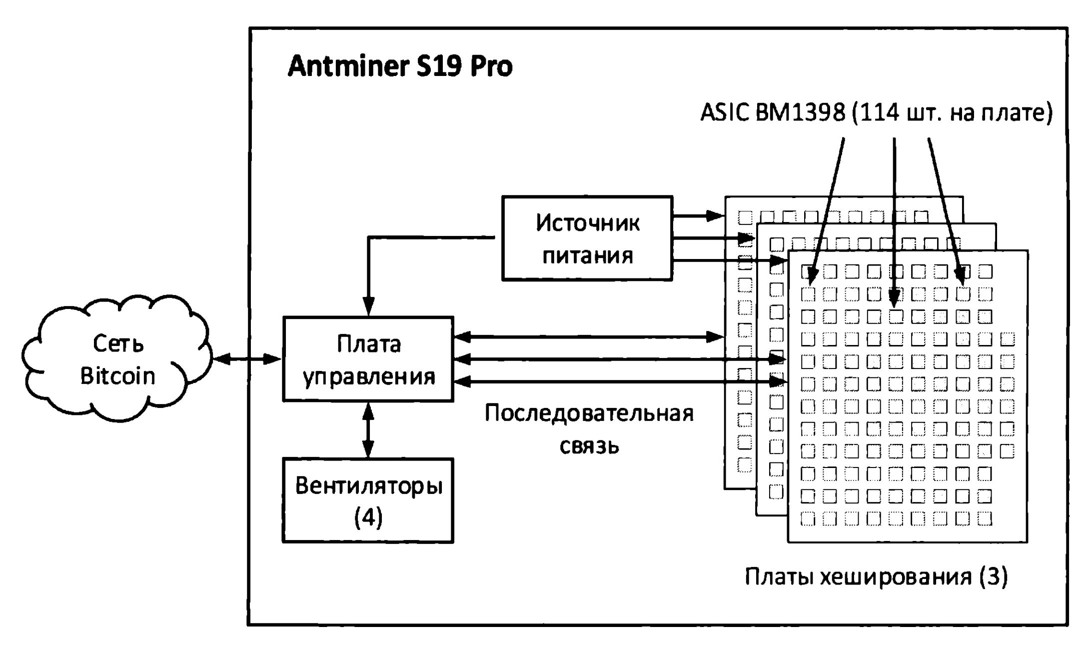


Рис. 15.3. Конфигурация аппаратных средств Antminer S19 Pro

При скорости хеширования 110 Тх/с уравнение (15.5) дает следующий средний интервал решения успешного добавления блока для Antminer S19 Pro:

$$T_B = \frac{10\ \text{минут}}{\left[\dfrac{110\times10^{12}}{140\times10^{18}}\right]} = 1{,}27\times10^{7}\ \text{минут}. \qquad (15.5)$$

Со средним интервалом успешного добавления блока в $1{,}27\times10^{7}$ минут ожидается, что при типичной для 2021 г. скорости хеширования Antminer S19 Pro сможет успешно добавлять блок в блокчейн каждые 24,2 года. И в этом случае данная конфигурация не слишком хорошо подходит для майнера-одиночки, и именно по этой причине все серьезные майнеры, располагающие небольшими ресурсами, присоединяются к майнинговым пулам, чтобы получать надежный, хотя и небольшой, доход.

Но что, если вы не занимаетесь мелким бизнесом, а вместо этого располагаете средствами для работы в промышленных масштабах? Допустим, вы покупаете 1000 машин Antminer S19 Pro и устанавливаете их в специальном помещении для компьютеров с соответствующим питанием, кондиционированием воздуха, контролем влажности и фильтрацией воздуха. Такой комплекс обеспечит средний интервал добавления блока, показанный в уравнении (15.6).

$$T_B = \frac{10\ \text{минут}}{\left[\dfrac{1000\times110\times10^{12}}{140\times10^{18}}\right]} = 1{,}27\times10^{4}\ \text{минут}. \qquad (15.6)$$

Эта рабочая конфигурация сокращает средний интервал добавления блока до $1{,}27\times10^{4}$ минут, что составляет около 8,8 суток. Такая конфигурация уже выглядит

более приемлемой для майнера, заинтересованного в получении более-менее регулярной отдачи от инвестиций в систему майнинга.

И что это за инвестиции — 15 млн долларов только за 1000 систем Antminer S19 Pro (если не учитывать возможную скидку на количество)! В следующем разделе будет рассмотрена экономика майнинга биткоинов с точки зрения первоначальных инвестиций, текущих расходов и ожидаемой доходности.

Экономика майнинга биткоинов

Помимо подключения к сети успешный майнинг биткоинов в промышленных масштабах требует четырех основных компонентов:

- подходящее помещение;
- оборудование для майнинга;
- электроэнергия;
- время.

Наше обсуждение до этого момента было сосредоточено на требованиях к обработке для решения задачи вычисления хеш-значений, необходимой для проверки биткоин-блока с целью его добавления в блокчейн. По мере того как оборудование для майнинга биткоинов становилось все более специализированным и мощным, общее количество потребляемой для майнинга электроэнергии неуклонно росло. Согласно анализу, проведенному Кембриджским университетом в феврале 2021 г., на майнинг биткоинов во всем мире расходуется в среднем больше электроэнергии, чем потребляет Аргентина.

Система Antminer S19 Pro, рассмотренная в предыдущем разделе, потребляет 3250 Вт , что эквивалентно 78 кВт-ч/сутки. В зависимости от цены на электроэнергию в регионе, где осуществляется майнинг, расходы на нее могут свести на нет большую часть или даже всю прибыль, которую можно было бы получить от успешного решения задачи вычисления хешей для блоков.

Тенденция в оборудовании для майнинга биткоинов заключалась в увеличении мощности хеширования каждого нового поколения систем майнинга при одновременном сокращении количества электроэнергии, необходимой для вычисления каждого хеша.

Это привело к разработке спецификаций для систем майнинга, которые количественно определяют мощность, потребляемую на операцию хеширования. Один джоуль (Дж) равен одному ватту мощности за период в одну секунду. Другими словами, 1 Вт = 1 Дж/с. Для определения коэффициента полезного действия системы Antminer S19 Pro следует поделить 3250 Дж/с на 110 Тх/с, что в итоге дает 29,5 Дж/Тх. Antminer S19 Pro, помимо обеспечения очень высокой скорости хеширования в 110 Тх/с, является одной из самых энергоэффективных систем майнинга биткоинов, доступных на рынке.

Несмотря на тенденцию к повышению эффективности хеширования, общее энергопотребление сети Bitcoin продолжает расти. Огромное потребление электроэнергии сетью рассматривается некоторыми как расточительство и как вклад в экологические и климатические проблемы.

Помимо желания уменьшить негативное влияние высокого энергопотребления при майнинге биткоинов, мелкие майнеры были недовольны тем, что их лишили воз-

можности добывать биткоины с разумной прибылью крупные предприятия промышленного майнинга, расположенные в странах по всему миру.

Биткоин далеко не единственная существующая в настоящее время криптовалюта. В 2021 г. в активно использовались более 10 000 различных криптовалют. Некоторые из них достигли такого уровня популярности и общей рыночной стоимости, что смогли в определенной степени конкурировать с биткоином. Некоторые из этих валют имеют специфические особенности для борьбы с предполагаемыми негативными тенденциями, характерными для биткоина. Мы обсудим некоторые из этих альтернативных криптовалют в следующем разделе.

Альтернативные виды криптовалют

Майнинг биткоинов начался как прибыльное развлечение для компьютерных энтузиастов, которые использовали свободные вычислительные циклы на своих процессорах. Поскольку применение дорогих, шумных и энергозатратных систем майнинга выросло до промышленных масштабов, возможность получать даже минимальную прибыль с помощью самодельной системы майнинга биткоинов исчезла.

Это одна из причин разработки многочисленных криптовалют в качестве альтернативы биткоину, которые обычно называют **альткоинами**. Некоторые альткоины нацелены на то, чтобы усложнить и удорожить разработку ASIC для решения необходимой для их майнинга задачи. Часть из них предназначена для того, чтобы избежать трудоемкого получения доказательства работы, которое составляет основу биткоина. Избегая намеренно интенсивных вычислительных операций, эти валюты существенно сокращают количество электроэнергии, необходимой для майнинга и совершения транзакций с ними.

Для того чтобы любая криптовалюта получила широкое признание и применение, новые пользователи должны быть уверены, что любые средства, которые они доверяют валюте, сохранят свою ценность с течением времени, а негативные последствия, такие как обнаружение того, что чей-то цифровой кошелек был опустошен вором, маловероятны. Это жесткие стандарты, которым должна соответствовать новая криптовалюта. Тем не менее несколько альткоинов, основанных на технологии блокчейна, достигли уровня широкого признания и применения. Далее описаны некоторые из основных альтернативных криптовалют на 2021 г. и их ключевые особенности.

- **Ethereum (Эфириум).** Это децентрализованная программная платформа, подобная платформе Bitcoin. Она обеспечивает поддержку смарт-контрактов и распределенных приложений, которые можно использовать для выполнения таких функций, как перевод платежей от покупателя продавцу. **Смарт-контракты** — это хранящиеся в блокчейне Ethereum программы, которые выполняются при удовлетворении предопределенных условий. Смарт-контракты предназначены для облегчения исполнения юридических соглашений открытым и поддающимся проверке способом. Криптовалюта Ethereum, используемая для оплаты вычислительных ресурсов и транзакций в сети Ethereum, носит название Ether (эфир). Вместо дорогостоящей в отно-

шении вычислительных ресурсов концепции получения доказательства работы, реализованной в Bitcoin, Ethereum использует процесс, называемый доказательством владения. **Доказательство владения** (proof of stake) основано на количестве монет Ether, которыми владеет майнер, и владение ими позволяет майнеру проверять транзакции в сети.

- **Litecoin.** Система Litecoin была запущена в 2011 г. как ответвление (форк) системы Bitcoin и имеет с ней много общего. Litecoin была разработана с акцентом на быстром одобрении транзакций даже при больших объемах транзакций. Время добавления блока в Litecoin было сокращено до 150 секунд по сравнению с 10 минутами в Bitcoin. Может быть создано до 84 млн лайткоинов, что превышает общее количество биткоинов.
- **Dogecoin.** Криптовалюта Dogecoin была создана в 2013 г., по-видимому, в качестве шутки, задуманной как заявление о продолжающихся спекуляциях с криптовалютами, такими как Bitcoin. Логотипом Dogecoin служит фотография собаки породы сиба-ину. В мае 2021 г. рыночная капитализация Dogecoin составляла 85 млрд долларов. Dogecoin является форком базы кода Litecoin и отличается от многих других криптовалют тем, что не имеет фиксированного ограничения на предложение валюты. Вместо этого в ней предусмотрен стабильный "уровень инфляции" в 5 млн новых монет Dogecoin, создаваемых в год. Время добавления блока для Dogecoin составляет 60 секунд.
- **Bitcoin Cash** — это форк Bitcoin. Bitcoin Cash была создана после возникновения разногласий между фракциями сторонников Bitcoin относительно предлагаемого обновления, которое позволило бы увеличить размеры блоков для наборов транзакций. Больший размер блока позволяет включать больше транзакций в один блок блокчейна. Это означает возможное сокращение времени ожидания выполнения транзакции. Комиссия за транзакции в Bitcoin Cash так же, как правило, ниже. Как и в случае с другими перечисленными здесь альтернативными криптовалютами, Bitcoin Cash использует отдельный от Bitcoin блокчейн.

Это лишь некоторые из наиболее известных альткоинов, используемых в настоящее время. Несмотря на тысячи существующих альтернативных криптовалют, Bitcoin остается доминирующей криптовалютой с точки зрения широты использования и стоимости транзакций. В ноябре 2021 г. рыночная капитализация Bitcoin составляла 1 трлн долларов.

Резюме

Эта глава началась с краткого введения в концепции, связанные с блокчейном, открытым криптографически защищенным реестром, содержащим последовательность финансовых транзакций. Обсуждение продолжилось обзором процесса майнинга биткоинов, который добавляет транзакции в последовательность блокчейна и вознаграждает тех, кто выполняет эту задачу, оплатой в биткоинах. Для майнинга биткоинов требуется высокопроизводительное вычислительное оборудование, ко-

торое часто специально разрабатывают для решения этой задачи. Глава завершилась описанием особенностей этих аппаратных архитектур.

Прочитав эту главу, вы получили представление о том, что такое блокчейн Bitcoin и как используется эта технология. Вы ознакомились с этапами процесса майнинга биткоинов и с ключевыми особенностями компьютерных архитектур для майнинга биткоинов.

В следующей главе будут рассмотрены возможности вычислительных архитектур для самоуправляемых автомобилей, включая типы датчиков и источники информации, которые предоставляют таким автомобилям входные данные во время движения, а также типы вычислений, необходимых для управления автомобилем в реальных дорожных ситуациях.

Упражнения

1. Откройте веб-страницу для просмотра информации о транзакциях блокчейна по адресу **https://bitaps.com** и найдите список последних блоков. Щелкните на номере блока, откроется окно с заголовком блока в шестнадцатеричном формате и хешем SHA-256. Скопируйте оба эти элемента и напишите программу, чтобы определить, является ли предоставленный хеш правильным хешем заголовка. Не забудьте дважды выполнить алгоритм SHA-256 для вычисления хеша заголовка.
2. Настройте полный равноправный узел Bitcoin и подключите его к сети Bitcoin. Загрузите программное обеспечение Bitcoin Core по адресу **https://bitcoin.org/en/download**. Вам потребуется быстрое подключение к Интернету и не менее 200 Гбайт свободного места на диске.

16

Архитектуры для самоуправляемых автомобилей

В этой главе описываются возможности вычислительных архитектур для самоуправляемых автомобилей. Глава начинается с обсуждения типов датчиков и источников информации, которые предоставляют таким автомобилям входные данные во время движения. Далее мы перейдем к обсуждению требований по обеспечению безопасности самоуправляемого автомобиля и его пассажиров, а также других транспортных средств, пешеходов и стационарных объектов. Затем будет приведено описание типов вычислений, необходимых для эффективного управления автомобилем. Глава завершится обзором примера архитектуры компьютера для самоуправляемого автомобиля.

Эта глава поможет вам ознакомиться с основами вычислительной архитектуры для самоуправляемых автомобилей и с типами датчиков, используемых такими транспортными средствами. Вы получите представление о типах вычислений, выполняемых самоуправляемыми автомобилями, и о проблемах безопасности, связанных с такими транспортными средствами.

В этой главе будут представлены следующие темы:

- обзор самоуправляемых автомобилей;
- аспекты безопасности самоуправляемых автомобилей;
- требования к аппаратным средствам и программному обеспечению для самоуправляемых автомобилей;
- вычислительная архитектура автономного транспортного средства.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Обзор самоуправляемых автомобилей

Несколько крупных производителей автомобилей и технологических компаний активно занимаются разработкой и продажей полностью самоуправляемых, или автономных, автомобилей. Утопическое представление о безопасных, полностью самоуправляемых транспортных средствах манит нас в будущее, в котором пассажиры могут отдыхать, читать или даже спать в пути, а вероятность попасть в серьезное дорожно-транспортное происшествие будет намного ниже по сравнению с достаточно опасной ситуацией в этой сфере в настоящее время.

Эта картина будущего прекрасна, однако нынешнее положение дел в области самоуправляемых автомобилей остается далеким от этой цели. Профильные эксперты прогнозируют, что потребуются десятилетия, чтобы полностью разработать и внедрить технологии, которые смогут обеспечить широкую поддержку и использование полностью автономного транспорта.

Для того чтобы разобраться в требованиях к системам автономного вождения, мы начнем с действий, которые выполняет водитель-человек для управления современным автомобилем.

- **Выбор передачи.** Для простоты мы предполагаем, что автомобиль оснащен автоматической коробкой передач, которая позволяет водителю выбирать режимы *парковки*, *движения вперед* и *движения назад*.
- **Выбор направления движения (руление).** Это действие выполняется посредством рулевого колеса.
- **Педаль газа.** Нажатие этой педали ускоряет автомобиль в направлении, выбранном рукояткой переключения передач.
- **Педаль тормоза.** Независимо от того, движется автомобиль вперед или назад, нажатие педали тормоза замедляет и в конечном счете останавливает его.

Для достижения целей полностью автономного вождения технологии должны быть развиты до такой степени, чтобы управление всеми четырьмя этими действиями можно было доверить датчикам и вычислительным системам, которыми оснащено автоматизированное транспортное средство.

Для того чтобы понять текущее состояние технологий самоуправляемых автомобилей относительно цели полностью автономного вождения, полезно обратиться к шкале, которая определяет этапы перехода от автомобилей с исключительно ручным управлением к полностью автономной архитектуре. Это тема следующего раздела.

Уровни автономности вождения

Общество автомобильных инженеров (Society of Automotive Engineers, SAE) определило шесть уровней автоматизации управления автомобилем (см. **https://www.sae.org/standards/content/j3016_202104/**), которые охватывают диапазон от отсутствия какой-либо автоматизации до полностью автоматизированных транспортных средств, в которых нет водителя-человека. Были выделены следующие уровни.

- **Уровень 0 — автоматизация вождения отсутствует.** Это отправная точка, описывающая, как осуществляется управление автомобилями с момента их изобретения. На уровне 0 водитель несет ответственность за все аспекты эксплуатации автомобиля, включая достижение намеченного пункта назначения при обеспечении безопасности автомобиля, его пассажиров и всего, что находится за его пределами.

  Автомобили уровня 0 могут содержать функции безопасности, такие как предупреждение о возможном столкновении и автоматическое экстренное торможение. Эти функции отнесены к уровню 0, т. к. они не обеспечивают управление автомобилем в течение длительного времени.

- **Уровень 1 — система помощи водителю.** На этом уровне автоматизированная система вождения может выполнять управление рулением либо ускорением/замедлением в течение продолжительного периода времени, но не обеими этими функциями одновременно. При использовании системы помощи водителю уровня 1 водитель должен непрерывно выполнять все функции вождения, кроме единственной автоматической функции. На уровне 1 функцию управления рулением называют **удержанием полосы движения** (lane-keeping assistance, LKA). Функцию управления ускорением/замедлением на уровне 1 называют **адаптивным круиз-контролем** (adaptive cruise control, ACC). При использовании функции помощи водителю этого уровня водителю необходимо постоянно контролировать обстановку, и он должен быть готов взять на себя управление автомобилем.

- **Уровень 2 — частичная автоматизация вождения.** Системы автоматизации вождения уровня 2 развивают возможности уровня 1, обеспечивая одновременное управление рулением и ускорением/замедлением. Как и на уровне 1, водителю необходимо постоянно контролировать обстановку и в любой момент быть готовым взять на себя управление автомобилем.

- **Уровень 3 — условная автоматизация вождения.** Система автоматизации вождения уровня 3 может выполнять все задачи вождения в течение продолжительных периодов времени. Водитель всегда должен присутствовать за рулем и быть готовым взять управление на себя, если автоматизированная система вождения отправит запрос на вмешательство человека. Основное различие между уровнями 2 и 3 заключается в том, что на уровне 3 от водителя не требуется постоянно контролировать работу автоматизированной системы вождения или поддерживать контроль над ситуацией за пределами

транспортного средства. Вместо этого водитель-человек должен быть готов в любой момент отреагировать на запрос о вмешательстве от автоматизированной системы вождения.

- **Уровень 4 — высокая автоматизация вождения.** Автоматизированная система вождения этого уровня может выполнять все задачи вождения в течение продолжительных периодов времени, а также способна автоматически реагировать на неожиданные условия таким образом, чтобы свести к минимуму риск для автомобиля, пассажиров и других лиц за его пределами. Процесс минимизации рисков называют *резервным режимом задачи управления движением* (driving task fallback), он может включать в себя действия, которые позволяют избежать риска и возобновить нормальное вождение или выполнить другие маневры, такие как остановка автомобиля в безопасном месте. Водитель-человек может взять на себя управление автомобилем, реагируя на переход в резервный режим, или в другое время, если это необходимо. Однако, в отличие от более низких уровней автоматизации вождения, в автомобиле уровня 4 не требуется наличие человека, готового взять на себя управление во время движения. Также не требуется, чтобы автомобили этого уровня автоматизации были оснащены органами управления для водителей-людей. Основная область применения систем автоматизации вождения 4-го уровня — такси и общественный транспорт, где эксплуатация транспортного средства ограничена определенным географическим регионом и известным набором дорог.
- **Уровень 5 — полная автоматизация вождения.** В транспортном средстве уровня 5 все задачи вождения всегда выполняет автоматизированная система вождения. Нет необходимости в органах управления, с помощью которых водитель-человек мог бы управлять транспортным средством, — все люди в нем являются пассажирами. Система вождения этого уровня должна обеспечивать управление транспортным средством на дорогах любого типа, в любое время суток и при любых погодных условиях, при которых ответственный и, как правило, опытный водитель-человек мог бы безопасно управлять автомобилем. Единственная задача, связанная с вождением, которую выполняют люди, находящиеся в автомобиле, — это выбор пункта назначения.

В 2021 г. большинство транспортных средств на дорогах относились к уровню 0. Многие новые модели автомобилей оснащены функциями автоматизации уровней 1 и 2. Во всем мире крайне мало одобренных для эксплуатации систем уровня 3. Для систем 3-го уровня, получивших одобрение регулирующих органов, обычно устанавливаются ограничения, определяющие конкретные условия эксплуатации, такие как вождение в условиях интенсивного движения на автомагистралях.

Важным отличием в требованиях к производительности вычислительных систем для автономных транспортных средств по сравнению со многими традиционными вычислительными устройствами, такими как смартфоны и веб-серверы, является вполне реальная опасность травмирования и смерти пассажиров автомобиля и других лиц за его пределами. Это тема следующего раздела.

Аспекты безопасности самоуправляемых автомобилей

В любое время, когда в движущемся автомобиле активен некоторый уровень автономного управления, алгоритмы, лежащие в основе автономного поведения, должны постоянно применять иерархический набор требований для удовлетворения потребностей пассажиров, принимая при этом все доступные меры, чтобы избежать негативных последствий, таких как столкновения с другими объектами.

Наивысшим приоритетом, закодированным в алгоритмах автономного транспортного средства, всегда должно быть обеспечение безопасности самого транспортного средства, его пассажиров и других лиц, находящихся поблизости. Рассмотрим альтернативу: если в качестве наивысшего приоритета выбрать доставку пассажиров в требуемый пункт назначения, то самоуправляемый автомобиль может интерпретировать это как разрешение проезда на красный свет и наезда на пешеходов, если эти действия обеспечат следование по самому быстрому пути к месту назначения.

Такой автомобиль должен не только прогнозировать дорожную обстановку и управлять своим движением с учетом многочисленных препятствий, он также должен гарантировать, что все его критически важные для безопасности компоненты работают должным образом и получают достоверные входные данные. Это означает, что если эффективность работы наиболее важных датчиков, например видеокамер, снижается из-за скопления снега или грязи, то алгоритмы автомобиля должны привести систему в состояние минимального риска. В такой ситуации автомобиль может принять меры для обеспечения безопасности путем уведомления водителя о необходимости взять управление на себя или выполнения остановки в безопасном месте.

Поскольку безопасное поведение автономных транспортных средств крайне важно, государственные регулирующие органы, ответственные за безопасность дорожного движения, требуют получения одобрения для их эксплуатации на дорогах общего пользования. Как правило, отдельный разработчик или компания не могут создать автономное транспортное средство и разрешить его эксплуатацию на дорогах без такого одобрения. Если кто-либо сделает это, и транспортное средство станет причиной аварии или травмы, юридическая ответственность за это происшествие может быть возложена на находившихся в нем людей или на сторону, разрешившую его эксплуатацию на дорогах.

Различные научные и коммерческие организации разработали ряд технологий, необходимых для автономного вождения, но проект полнофункционального и полностью автономного транспортного средства пока еще не представлен. Следующие четыре этапа дают приблизительное представление о возможностях автономного вождения, которые были продемонстрированы и потребуются для достижения 5-го уровня.

- **Этап 1. Следование дороге.** Автоматизированная система вождения, обеспечивающая следование дороге, способна обнаруживать дорожную разметку,

включая линии, разграничивающие полосы движения, и следовать ей, а также может обнаруживать изменение текстуры покрытия проезжей части и обочины на дорогах без разметки. Система вождения, которая лишь удерживает автомобиль в пределах полосы движения, не выполняет таких необходимых для вождения функций, как соблюдение правил дорожного движения и объезд других транспортных средств.

- **Этап 2. Соблюдение правил дорожного движения.** На этом этапе система автоматизации вождения может удерживать автомобиль в пределах полосы движения, а также обнаруживать и выполнять указания дорожных знаков и светофоров по разрешенным направлениям движения. Система с таким уровнем возможностей способна надежно реагировать на дорожные знаки, предоставляющие информацию для вождения, такую как ограничение скорости или требование уступить дорогу на перекрестке.
- **Этап 3. Избегание препятствий.** Система вождения, способная избегать препятствий, удерживает автомобиль в полосе движения и обеспечивает выполнение требований сигналов регулирования дорожного движения, а также обнаруживает все значимые движущиеся или неподвижные объекты вблизи автомобиля и реагирует соответствующим образом, чтобы свести к минимуму риск для всех. К препятствиям относятся другие транспортные средства, велосипедисты, пешеходы, животные, дорожные сооружения, мусор на проезжей части и другие помехи движению, возникающие в необычных ситуациях, например затопленные или размытые участки дороги.
- **Этап 4. Надежное разрешение пограничных ситуаций.** Может показаться, что автоматизированная система вождения, которая справляется с первыми тремя этапами, перечисленными выше, является достаточно качественной для разрешения ее эксплуатации, однако самый сложный аспект автоматизированного вождения заключается в способности системы должным образом справляться с редкими, но важными ситуациями, в которых от водителя-человека можно ожидать надлежащей реакции. Например, предположим, что наводнение привело к обрушению части моста в реку.

  Движущийся по мосту автомобиль, управляемый человеком, в идеальном случае должен остановиться, когда водитель заметит, что часть моста отсутствует. Для того чтобы доверять возможностям автоматизированной системы вождения, пассажиры автономного транспортного средства должны быть уверены, что она будет адекватно реагировать во всех ситуациях, в которых компетентный водитель-человек сможет эффективно минимизировать риски для всех людей внутри и снаружи автомобиля.

  Более будничным примером таких возможностей является ситуация, когда два автомобиля почти одновременно подъезжают к равнозначному четырехстороннему перекрестку с разных направлений. Один водитель может подать другому знак рукой, чтобы тот проехал перекресток. Если второго водителя замещает автономная система, обнаружит ли она такой жест и отреагирует ли на него соответствующим образом?

Достижение способности реагировать на редкие, но потенциально опасные события на этапе 4 может показаться логическим продолжением возможностей, описанных для первых трех этапов, однако этот сценарий на самом деле представляет собой серьезную проблему для автоматизированной системы вождения. Как мы увидим позже в этой главе, нейронные сети являются основной технологией, используемой в настоящее время для создания автоматизированных систем вождения. Нейронные сети учатся на серии примеров ситуаций, представленных им вместе с "правильным" ответом, который сеть должна выдавать в каждой ситуации. Эти сети продемонстрировали потрясающую способность делать выводы из представленных им ситуаций и правильно реагировать на новые ситуации, которые укладываются в рамки учебных сценариев, изученных сетью.

Процесс работы с ситуациями, которые в определенном смысле находятся "между" сценариями обучения сети, называют **интерполяцией**. Проблемы возникают, когда нейронные сети пытаются перенести обобщения на сценарии, которые выходят за рамки сценариев их обучения. Этот процесс называют **экстраполяцией**. Основанная на экстраполяции реакция нейронной сети на новую ситуацию может соответствовать, а может и не соответствовать тому, что посчитал бы правильным компетентный водитель-человек. Эффективное решение очень большого числа редких, но возможных сценариев вождения, с которыми водители-люди сталкиваются ежедневно, может быть самой большой проблемой, стоящей перед разработкой систем автономного вождения.

В следующем разделе представлены входные данные, предоставляемые системам автономного вождения различными датчиками, установленными в транспортном средстве.

Требования к аппаратным средствам и программному обеспечению для самоуправляемых автомобилей

Водители-люди должны контролировать состояние своих транспортных средств и постоянно оценивать окружающую обстановку, отслеживая движущиеся и неподвижные препятствия. Основным средством сбора этой информации является зрение.

Используя зрение, опытный водитель следит за приборами своего автомобиля, главным образом за спидометром, и наблюдает за окружающей обстановкой, чтобы удерживать автомобиль в полосе движения, соблюдать безопасное расстояние до других транспортных средств, требования дорожных знаков и светофоров, а также избегать любых препятствий на поверхности проезжей части или вблизи нее.

Водители-люди в меньшей степени полагаются на другие органы чувств, включая использование слуха для обнаружения таких сигналов, как автомобильные гудки и сигнализация на железнодорожных переездах. Чувство осязания также вступает в игру, когда, например, на поверхности проезжей части устанавливаются шумовые полосы, предупреждающие о приближении к перекрестку. Осязание также может

помочь, когда невнимательный водитель съезжает с проезжей части на обочину, текстура которой обычно существенно отличается от поверхности проезжей части дороги.

Зрение, слух и осязание — это исчерпывающий список каналов получения входной информации, которые водитель-человек использует во время вождения. С одной стороны, поступающая от этих органов чувств информация позволяет водителям-людям преодолевать, в основном успешно, миллиарды миль каждый день. С другой, очевидно, что в процессах распознавания опасных ситуаций и принятия надлежащих мер реагирования на них сохраняются значительные пробелы, о чем свидетельствуют тысячи смертей, ежедневно происходящих вследствие дорожно-транспортных происшествий.

Для того чтобы водители приняли системы автономного вождения, эти системы обязаны быть не просто такими же безопасными, как водители-люди, — они должны намного превосходить их по таким показателям, как количество дорожно-транспортных происшествий на пройденный километр. Такой высокий уровень рабочих характеристик будет необходим, т. к. многие водители-люди будут неохотно передавать управление автоматизированной системе, пока не убедятся, что она превосходит их собственный (возможно, надуманный) уровень водительских навыков.

В следующем разделе представлены типы датчиков, используемых в системах автономного вождения, и их вклад в достижение целей по безопасности и надежности.

Наблюдение за состоянием транспортного средства и его окружением

Датчики, используемые для автономного вождения, должны точно оценивать состояние самого транспортного средства, а также положение и скорость (в данном случае **скорость** — это сочетание скорости и направления движения) всех значимых объектов, находящихся вблизи транспортного средства. В следующих разделах описываются основные типы датчиков, используемых в конструкциях современных автономных транспортных средств.

GPS, спидометр и инерциальные датчики

Автономное транспортное средство должно постоянно контролировать свое состояние, включая местоположение, направление движения и скорость. Информация о местоположении транспортного средства используется как для низкоуровневых задач вождения, таких как удержание полосы движения, так и для функций более высокого уровня, таких как построение маршрута движения к месту назначения.

Для определения местоположения на низком уровне можно использовать данные лидара или изображения с видеокамер, которые предоставляют подробную информацию, вплоть до положения автомобиля относительно линий разметки полосы движения. Эти данные имеют высокое разрешение (точность доходит до сантиметров) и обновляются с высокой скоростью (возможно, десятки или даже сотни раз в

секунду), что обеспечивает плавную и непрерывную реакцию системы автономного вождения на изменяющиеся условия.

Данные, предоставляемые приемником **глобальной системы позиционирования** (global positioning system, GPS), обновляются реже (возможно, несколько раз в секунду) и могут иметь гораздо меньшую точность — погрешность определения местоположения способна достигать нескольких метров. Информация, предоставляемая GPS-приемником, идеально подходит для планирования маршрута, но точность таких измерений невелика и обновляются они слишком редко, чтобы быть полезными для удержания автомобиля в полосе движения.

Приемники GPS имеют существенное ограничение работоспособности, поскольку в своей работе они полагаются на непрерывный прием спутниковых сигналов. В ситуациях, когда видимость неба из автомобиля ухудшается, например на дороге в густом лесу, на узкой улице среди высоких домов в центре города или в тоннеле, GPS-приемник может вообще не работать.

Спидометр обычно обеспечивает точное измерение скорости транспортного средства на основе скорости вращения колес. Иногда показания спидометра могут неточно отражать скорость автомобиля вследствие пробуксовки колес из-за наличия на проезжей части грязи или льда. В качестве дублера спидометра можно использовать GPS-приемник, который достаточно точно измеряет скорость автомобиля, если получает необходимые спутниковые сигналы. В этой ситуации расхождение между показаниями спидометра автомобиля и скоростью, измеренной GPS-приемником, может свидетельствовать о пробуксовке колес, что является небезопасным явлением.

Современные транспортные средства часто содержат инерциальные датчики в виде акселерометров или, в некоторых случаях, гироскопа. Акселерометр измеряет ускорение, или темп изменения скорости, вдоль одной оси движения. Люди ощущают ускорение как силу, вдавливающую их в сиденье, когда автомобиль набирает скорость, и как силу, толкающую их вбок во время резкого поворота.

Автомобили обычно содержат два акселерометра: один для измерения ускорения и замедления вдоль продольной оси (вперед-назад), а второй — для измерения ускорения вдоль поперечной оси (из стороны в сторону). Акселерометр, измеряющий ускорение в поперечном направлении, позволяет оценивать инерционные эффекты поворотов. Гироскоп можно использовать для непосредственного измерения скорости поворота автомобиля.

Акселерометры могут выполнять измерения с очень высокой скоростью и используются для таких целей, как приведение в действие подушек безопасности в течение миллисекунд после обнаружения столкновения. Акселерометры и гироскоп в транспортном средстве отслеживают его ориентацию и скорость, позволяя осуществлять точный контроль его положения относительно окружения.

Используя алгоритм, называемый **фильтром Калмана**, можно комбинировать измерения из нескольких источников, например GPS-приемника и инерциальных датчиков, с математической моделью физических законов, которые ограничивают движение объектов, таких как автомобиль. Процесс калмановской фильтрации учи-

тывает, что каждое измерение содержит некоторую ошибку, а математическая модель является несовершенным представлением динамического поведения автомобиля. Учитывая характеристики статистической погрешности датчиков и диапазон непредсказуемого поведения, с которым может столкнуться транспортное средство (в результате ускорения, торможения и поворота руля, а также внешних воздействий, таких как движение на подъем и под уклон), фильтр Калмана синтезирует доступную информацию для получения оценки состояния транспортного средства, которое является значительно более точным, чем информация, полученная от любого датчика или математической модели самих по себе.

GPS-приемник, спидометр и инерциальные датчики дают оценку состояния самого транспортного средства. Для безопасного управления им и достижения требуемого пункта назначения автономная система вождения также должна воспринимать окружающую транспортное средство среду. В следующих разделах рассматриваются датчики, выполняющие эту функцию.

Видеокамеры

Некоторые системы автономного вождения в качестве основного датчика окружающей обстановки используют множество видеокамер. Применяемые в таких автомобилях видеокамеры имеют функции, знакомые пользователям цифровых видеокамер потребительского класса, включая портативные устройства и видеокамеры в смартфонах. Видеокамера автономного транспортного средства — это устройство с умеренно высоким разрешением, обычно 1920 × 1080 пикселов. Автомобиль может быть оснащен несколькими расположенными по его периметру камерами с перекрывающимися полями обзора.

Применение множества камер подчеркивает одно из преимуществ автономных транспортных средств по сравнению с автомобилями, управляемыми человеком: установленные в них системы технического зрения способны одновременно и непрерывно отслеживать ситуацию вокруг транспортного средства во всех направлениях. Водители-люди, для сравнения, могут в каждый момент времени смотреть только в одном направлении, располагая некоторым ограниченным периферийным зрением вокруг более крупного основного поля зрения. Наличие внутренних и внешних зеркал в транспортных средствах, управляемых человеком, помогает несколько компенсировать это неотъемлемое ограничение человеческого зрения, хотя обычно существуют довольно обширные слепые зоны, о которых водители-люди должны знать и которые должны учитывать при оценке ситуации.

Самая большая проблема, связанная с использованием видеокамер в системах автономного вождения, заключается в том, что видеоизображения, выводимые этими устройствами, не несут непосредственной пользы для выполнения задач вождения. Для того чтобы выделить на изображениях существенные признаки, объединить их для распознавания отдельных объектов, а затем понять, что делают эти объекты, и отреагировать соответствующим образом, необходимо выполнить серьезную обработку изображений. Мы обсудим обработку видеоизображений в системах автономного вождения в *разд. "Распознавание окружающей обстановки" далее в этой главе*.

Радар

Одно из ограничений видеокамер в системах автономного вождения заключается в том, что они формируют плоское двумерное изображение сцены, просматриваемой камерой. Без существенной дополнительной обработки невозможно определить, находится ли часть сцены, представленная каким-либо отдельным пикселом, близко к транспортному средству (и поэтому представляет собой потенциальное препятствие, требующее немедленной реакции) или далеко от него и не имеет отношения к задаче вождения.

Частично решить эту проблему помогает технология **радиолокационного обнаружения и измерения дальности** (**ra**dio **d**etection **a**nd **r**anging, radar). Радарная система периодически отправляет импульсы электромагнитной энергии в окружающую среду и воспринимает отражения этих сигналов от объектов, находящихся поблизости.

Радар может обнаруживать объекты и определять направление на каждый объект, а также дальность до этих объектов. Он также способен измерять скорость объектов относительно транспортного средства, на котором установлена радарная система. Радар может отслеживать транспортное средство, находящееся впереди на расстоянии нескольких сотен метров, и помогать поддерживать безопасную дистанцию при использовании в составе системы адаптивного круиз-контроля (ACC).

Однако радарные системы имеют некоторые существенные ограничения. По сравнению с видеокамерой сцена, воспринимаемая типичным автомобильным радаром, намного более размыта и имеет меньшее разрешение. Радарные системы также подвержены помехам при измерениях, что снижает уверенность в качестве предоставляемой ими информации.

Несмотря на эти ограничения, радарные датчики работают без снижения эффективности в ситуациях, когда видеокамеры теряют способность нормально функционировать, например при сильном дожде, снеге и густом тумане.

Лидар

Вместо видеокамер некоторые разработчики автономных транспортных средств решили доверить решение задач определения местоположения и ориентации транспортного средства и обнаружения объектов вблизи него лазерным системам. Технология **лазерного обнаружения и измерения дальности** (**li**ght **d**etection **a**nd **r**anging, lidar) использует лазер для сканирования местности вокруг автомобиля и сбора данных посредством восприятия отражений от поверхностей и объектов.

Каждое измерение лидара представляет собой расстояние от лазерного устройства подсветки до поверхности, которая отражает часть лазерной энергии обратно на лидар. Направление лазерного луча относительно транспортного средства во время каждой серии измерений известно программному обеспечению лидара. Расстояние до точки отражения и обратно вычисляется с помощью измерения интервала времени между моментом испускания лазерного импульса и возвращением эхосигнала на лидар, который называется временем прохождения импульса. Расстояние, прой-

денное импульсом до цели и обратно, равно времени прохождения импульса, умноженному на скорость света в атмосфере.

Результатом быстрых последовательных измерений во всех направлениях вокруг транспортного средства является создание трехмерного набора точек, называемого **облаком точек**. Оно представляет расстояния от лидарного датчика до окружающих поверхностей, включая дорожное покрытие, здания, транспортные средства, деревья и другие объекты.

Сонар

Технология **обнаружения и определения дальности с помощью звуковых волн** (**so**und **n**avigation **a**nd **r**anging, sonar) выполняет функцию, аналогичную функциям радарных и лидарных систем, за исключением того, что сонары излучают звуковые импульсы, а не электромагнитные волны.

Сонары обычно работают на гораздо меньших расстояниях, чем радары и лидары, и в основном используются для обнаружения препятствий, находящихся в непосредственной близости от автомобиля, например при парковке между другими автомобилями или для обнаружения ситуаций, когда автомобиль, следующий по соседней полосе, приближается к вам на недопустимо близкое расстояние.

Распознавание окружающей обстановки

Поскольку датчики собирают необработанную информацию об автомобиле и его окружении, выходные данные датчика нельзя сразу же использовать для эффективного управления автомобилем. Необходимо выполнить несколько этапов обработки, чтобы преобразовать необработанные измерения датчиков в полезную информацию, которую можно использовать для автономного вождения. В следующих разделах описываются этапы обработки, необходимые для преобразования данных датчиков в решения об управлении автономными транспортными средствами, которые полагаются на видеокамеры и лидары в качестве основных датчиков. Мы начнем с обработки изображений с видеокамер с использованием сверточных нейронных сетей.

Сверточные нейронные сети

В *главе 6* были кратко представлены концепции глубокого обучения и искусственных нейронных сетей. Напомню, что искусственные нейроны — это математические модели, предназначенные для воспроизведения поведения биологических нейронов, которые являются клетками мозга, ответственными за восприятие и принятие решений.

В современных автономных транспортных средствах, которые используют видеокамеры для оценки окружающей обстановки, ведущей технологией извлечения информации для принятия решений из получаемых от камер видеоизображений является сверточная нейронная сеть.

Сверточная нейронная сеть (Convolutional neural network, CNN) — это специализированный тип искусственной нейронной сети, которая применяет определенную форму фильтрации, называемую сверткой, к данным необработанных изображений, чтобы извлечь информацию для обнаружения и идентификации объектов на изображениях.

Изображение, снятое видеокамерой, представляет собой прямоугольную сетку пикселов. Цвет каждого пиксела представлен набором значений интенсивности красного, зеленого и синего цветов. Каждое из этих значений интенсивности представляет собой 8-битное целое число в диапазоне от 0 до 255. Интенсивность 0 означает, что соответствующий цвет отсутствует в цвете пиксела, в то время как 255 представляет максимальную интенсивность данного цвета. Цвет пиксела представлен тремя значениями для красного, зеленого и синего цветов, сокращенно обозначаемых **RGB**. Некоторые примеры цветов RGB представлены в табл. 16.1.

Таблица 16.1. *Примеры цветов в формате RGB*

Интенсивность красного цвета	Интенсивность зеленого цвета	Интенсивность синего цвета	Цвет
255	0	0	Красный
0	255	0	Зеленый
0	0	255	Синий
0	0	0	Черный
255	255	255	Белый
128	128	128	Серый
255	255	0	Желтый
255	0	255	Пурпурный
0	255	255	Светло-синий

Сеть CNN получает в качестве входных данных снятое видеокамерой RGB-изображение в виде трех отдельных двумерных массивов 8-битных значений (цветов пикселов) для красного, синего и зеленого цветов. Мы будем называть их тремя **цветовыми плоскостями** изображения.

Затем CNN выполняет сверточную фильтрацию для каждой из цветовых плоскостей. Сверточный фильтр состоит из прямоугольной сетки чисел, которая обычно имеет небольшой размер, например две строки на два столбца. Операция свертки начинается с левого верхнего угла каждой цветовой плоскости. С математической точки зрения свертка выполняется путем умножения интенсивности цвета в позиции каждого пиксела на число в соответствующем месте сверточного фильтра и последующего суммирования результатов этих операций по всем элементам

фильтра. Результатом этой операции является один элемент, полученный в итоге свертки. Вскоре мы рассмотрим пример, который поможет прояснить этот процесс.

Свертка продолжается смещением фильтра на один или более пикселов вправо с последующим повторением операций умножения и суммирования с пикселами изображения, охваченными новым местоположением фильтра. Эти операции приводят к получению результата свертки для второй позиции в верхней строке. Данный процесс продолжается по всей верхней строке изображения, после чего фильтр смещается обратно к левому краю и перемещается на один или более пикселов вниз, и процесс повторяется для второй строки изображения. Эта последовательность смещения фильтра, умножения элементов фильтра на интенсивность цвета пикселов в местоположении фильтра и суммирования результатов продолжается по всему изображению.

Пример реализации сверточной нейронной сети

Теперь мы разберем простой пример, чтобы продемонстрировать, как работают сети CNN. Допустим, у нас есть изображение размером 5×5 пикселов и сверточный фильтр размером 2×2 пиксела. Мы рассмотрим только одну из трех цветовых плоскостей, хотя в реальных фильтрах одни и те же операции будут выполняться отдельно для всех трех цветовых плоскостей.

Мы будем сдвигать фильтр на один пиксел вправо и на один пиксел вниз каждый раз, когда потребуется такое перемещение. Расстояние (в пикселах), на которое сверточный фильтр сдвигается при каждом перемещении, называется **шагом по индексу**. Для простоты все пикселы и элементы сверточного фильтра в этом примере будут представлены в виде однозначных целых чисел. В табл. 16.2 приведен пример данных нашей цветовой плоскости.

Таблица 16.2. *Пример данных цветовой плоскости*

1	4	6	7	8
2	9	2	0	5
8	2	1	4	7
3	9	0	6	8
2	1	4	7	5

Табл. 16.3 содержит сверточный фильтр. Каждое число в таблице представляет собой один коэффициент фильтрации:

Таблица 16.3. *Пример сверточного фильтра*

7	2
8	0

Мы будем использовать шаг по индексу 1. Это означает, что для каждой последовательной оценки выходных данных фильтра будет выполняться сдвиг на 1 пиксел вправо, а затем на 1 пиксел вниз с возвратом к левому краю, когда будет достигнут правый край изображения.

Для первой оценки фильтра мы умножаем пикселы в левом верхнем углу изображения на соответствующие элементы из сверточного фильтра (табл. 16.4).

Таблица 16.4. *Шаг умножения при выполнении свертки*

1×7	4×2	6	7	8
2×8	9×0	2	0	5
8	2	1	4	7
3	9	0	6	8
2	1	4	7	5

После умножения интенсивностей цветовой плоскости на элементы фильтра свертки в сетке 2 × 2 мы складываем их вместе, как показано в уравнении (16.1).

$$C_{1,1} = (1 \times 7) + (4 \times 2) + (2 \times 8) + (9 \times 0) = 31. \qquad (16.1)$$

Результат поэлементного умножения интенсивностей цветовой плоскости на коэффициенты фильтра с последующим суммированием каждого из этих произведений является одним элементом нашего выходного массива, $C_{1,1}$. Этот результат мы помещаем в ячейку на пересечении строки 1 и столбца 1 выходного массива, как показано в табл. 16.5.

Таблица 16.5. *Первый выходной элемент свертки*

31			

Затем мы сдвигаем сетку фильтра на одну позицию вправо и снова выполняем операции умножения и суммирования, как показано в табл. 16.6.

Таблица 16.6. *Шаг умножения при выполнении свертки*

1	4×7	6×2	7	8
2	9×8	2×0	0	5
8	2	1	4	7
3	9	0	6	8
2	1	4	7	5

Результат умножения интенсивностей цветовой плоскости на элементы сверточного фильтра во второй позиции показан в уравнении (16.2).

$$C_{1,2} = (4 \times 7) + (6 \times 2) + (9 \times 8) + (2 \times 0) = 112. \quad (16.2)$$

В табл. 16.7 представлен результат свертки после ввода второго элемента в ячейку на пересечении строки 1 и столбца 2.

Таблица 16.7. *Второй выходной элемент свертки*

31	112		

Мы продолжаем смещать фильтр на один пиксел вправо для получения каждой оценки в верхней строке. Поскольку наш фильтр имеет ширину 2 пиксела, а изображение — 5 пикселов, мы можем разместить фильтр, не допуская его выхода за пределы изображения, только в четырех позициях. По этой причине выходной массив фильтра содержит 4 строки и 4 столбца.

После завершения расчетов для верхней строки изображения мы возвращаем фильтр к левому краю изображения и перемещаем его на один пиксел вниз, чтобы вычислить элемент в ячейке на пересечении строки 2 и столбца 1 ($C_{2,1}$). Эта процедура повторяется для всего изображения вплоть до ее завершения в правом нижнем углу. Итоговый результат операции свертки для нашего изображения размером 5×5 пикселов показан в табл. 16.8.

Таблица 16.8. *Результат операции свертки*

31	112	72	65
96	83	22	42
84	88	15	90
55	71	44	114

Это объяснение операции свертки могло показаться вам несколько утомительным и малополезным, но теперь мы переходим к хорошей части.

Сверточные нейронные сети в системах автономного вождения

Ниже перечислены наиболее интересные моменты, связанные с использованием сверточной фильтрации для обработки видеоизображений в системах автономного вождения.

- Математически свертка состоит из простых операций умножения и сложения.

* Сверточные нейронные сети (CNN) можно обучать таким же образом, как и **искусственные нейронные сети** (Artificial neural network, ANN), обсуждавшиеся в *главе 6*. Их обучение состоит из повторных испытаний, в которых нейронной сети предоставляют набор входных данных вместе с "правильным" выводом, который должен быть получен при вводе в нее этих данных. Процесс обучения корректирует веса внутри ANN (и в фильтрах свертки CNN), чтобы улучшить способность сети правильно реагировать на заданные входные данные на каждой итерации.
* В рамках процесса обучения CNN определяет наилучшие значения данных для размещения в каждой ячейке своих таблиц сверточных фильтров.

В качестве конкретного примера рассмотрим задачу управления транспортным средством, которая заключается в его удержании по центру занимаемой полосы движения. Общая процедура обучения CNN выполнению этой задачи заключается в записи серии видеофрагментов с камер автомобиля вместе с соответствующими сигналами рулевого управления, которые автономная система должна попытаться воспроизвести в качестве выходных данных.

Обучающий набор данных должен быть достаточно разнообразным, чтобы гарантировать, что изучаемое CNN поведение охватывает весь диапазон предполагаемых условий эксплуатации. Это означает, что данные для обучения должны включать в себя вождение по прямым дорогам и дорогам с различными изгибами. Эти данные также должны описывать вождение в дневное и ночное время и в других возможных условиях, такие как различные погодные условия.

Во время обучения каждый обучающий пример предоставляется сети в виде видеоизображения вместе с правильной реакцией рулевого управления. По мере того как CNN учится правильно реагировать на изменяющиеся входные данные, она корректирует коэффициенты в своих сверточных фильтрах для выявления особенностей изображения, которые полезны при выполнении требуемых задач.

Одной из функций обработки изображений, которая особенно полезна при автономном вождении, является **обнаружение краев**. Оно включает в себя определение местоположения разграничительных линий, которые отделяют различающиеся части изображения друг от друга. Например, линии разметки, нанесенные на проезжую часть, всегда сильно отличаются по цвету от дорожного покрытия, что делает их хорошо заметными как для водителей-людей, так и для автономных систем.

Наиболее интересное и даже захватывающее свойство процесса обучения сети CNN заключается в том, что в результате простой демонстрации CNN видеокадров поверхности дороги в сочетании с точными примерами требуемых выходных данных CNN будет обнаруживать полезные атрибуты изображения (например, края) и автоматически создавать сверточные фильтры, которые специально настроены для выявления полезных особенностей изображений.

Поскольку фильтр CNN применяется многократно по всему изображению, он может обнаруживать объекты в любом месте изображения, где бы они ни появлялись. Это важная особенность автономного вождения, т. к. место появления важного объекта на видеоизображении предсказать невозможно.

На практике CNN могут содержать несколько сверточных фильтров, каждый из которых содержит разные наборы коэффициентов. Каждый из этих фильтров настраивается в процессе обучения для поиска разных типов особенностей, результаты которого могут быть обработаны на более поздних этапах работы сети для выполнения функций более высокого уровня, таких как обнаружение знаков остановки, светофоров и пешеходов.

Полная структура CNN содержит один или несколько этапов свертки, каждый из которых работает так, как описано в этом разделе. На начальном этапе свертки выполняется фильтрация для обнаружения простых объектов, таких как границы, а на более поздних этапах эти объекты объединяются для обнаружения более сложных объектов, таких как знак остановки.

Для надежного обнаружения такого объекта, как знак "Стоп", в различных условиях обучающие данные должны содержать изображения этих знаков, видимых вблизи и издалека, а также при взгляде на них прямо и сбоку под углом, и все это при различном освещении и при разной погоде.

В структуре CNN элементы данных, вычисленные на каждом этапе свертки, пропускаются через функцию активации для получения конечного результата этапа. Для этой цели можно использовать множество функций, однако общепринятой функцией активации является **блок линейной ректификации**, обозначаемый сокращением **RELU** (Rectified Linear Unit).

Функция активации RELU представляет собой очень простую формулу: проверьте каждый элемент таблицы выходных значений операции свертки, такой как показана в табл. 16.8. Если элемент меньше 0, замените его на 0. Вот и всё, что требуется. Наш простой пример в табл. 16.8 не содержит отрицательных элементов, поэтому применение функции RELU его не изменит. На практике функция активации RELU дает важные преимущества, включая повышенную скорость обучения по сравнению с другими распространенными функциями активации.

Другим этапом, используемым в архитектурах CNN, является этап группировки. Изображения высокого разрешения содержат большое количество пикселов. Для того чтобы ограничить потребление памяти и требования к обработке разумными пределами, необходимо уменьшить количество проходящих через сеть данных, сохранив при этом характерные особенности, которые обеспечивают правильное распознавание объектов на изображении. Одним из способов решения этой задачи является подход, называемый группировкой. **Группировка** объединяет несколько элементов таблицы входных данных в один элемент выходной таблицы. Например, каждое подмножество 2×2 пиксела из табл. 16.8 можно сгруппировать, чтобы уменьшить размер выходных данных этого подмножества до одного числового значения.

Есть несколько способов группировки пикселов внутри подмножества для получения одного значения. Одна из очевидных возможностей заключается в вычислении среднего значения. Для области размером 2×2 пиксела в левом верхнем углу табл. 16.8 среднее значение равно $(31 + 112 + 96 + 83) / 4 = 80{,}5$. Усреднение дает представление, сочетающее в себе эффекты от всех пикселов в данной области, од-

нако этот метод группировки не зарекомендовал себя как обеспечивающий наилучшие показатели в CNN. Вместо этого во многих случаях было продемонстрировано, что хорошие показатели обеспечивает простой выбор в качестве результата группировки максимального значения в пределах области. В этом примере максимальное значение в выбранной области равно 112.

Этот подход называется **группировкой с выбором максимума**. В табл. 16.9 представлены результаты группировки с выбором максимума для области размером 2 × 2 к табл. 16.8.

Таблица 16.9. *Результат группировки с выбором максимума для данных из табл. 16.8*

112	72
88	114

Для того чтобы преобразовать слой из двумерной структуры видеоизображений и сверточных слоев, преобразующих видеоданные, в форму, подходящую для ввода в традиционную ANN, необходимо перевести эти данные в одномерный формат.

Как и в случае с другими математическими операциями, описанными ранее, это простая процедура. Преобразование двумерной структуры в одномерный вектор называется **сглаживанием**. В сглаживающем слое коэффициенты в двумерной структуре просто последовательно переносятся в вектор, который выступает в качестве входных данных для традиционной ANN.

В табл. 16.10 представлены результаты выполнения операции сглаживания над данными из табл. 16.9.

Таблица 16.10. *Результат сглаживания данных из табл. 16.9*

112	72	88	114

В табл. 16.10 представлен одномерный вектор числовых значений, который имеет подходящий формат для использования в качестве входных данных в традиционной ANN.

После этапов свертки, группировки и сглаживания CNN реализует один или несколько уровней ANN, каждый из которых образует скрытый слой нейронов. Эти скрытые слои обычно представляют собой полностью связанные наборы нейронов, как показано на рис. 6.5 в *главе 6*. За скрытым слоем следует выходной слой, который представляет конечные выходные данные сети для использования в дальнейшей обработке.

Когда достаточно представительный набор данных для обучения объединен со структурой CNN соответствующего размера, способной к обучению и сохранению необходимых знаний, появляется возможность закодировать в коэффициентах CNN

и ANN огромный объем информации, представляющей широкий спектр сценариев вождения.

При проектировании автономного транспортного средства, пригодного для эксплуатации на дорогах общего пользования, результатом процесса проектирования и обучения CNN является система, способная распознавать и идентифицировать типы объектов, встречающиеся во всем спектре дорожных ситуаций, с которыми ежедневно сталкиваются водители. Это чрезвычайно сложная задача, и может пройти несколько лет, прежде чем возможности вычислительных технологий и программного обеспечения (как в структуре сети, так и в процессе обучения) будут способны стабильно превосходить водителей-людей в задачах обнаружения и распознавания объектов.

На рис. 16.1 представлена базовая (соответствующая описанному выше этапу 1) архитектура CNN, которая показала свою способность управлять системой автономного вождения, чтобы удерживать автомобиль в занимаемой им полосе движения на извилистой дороге.

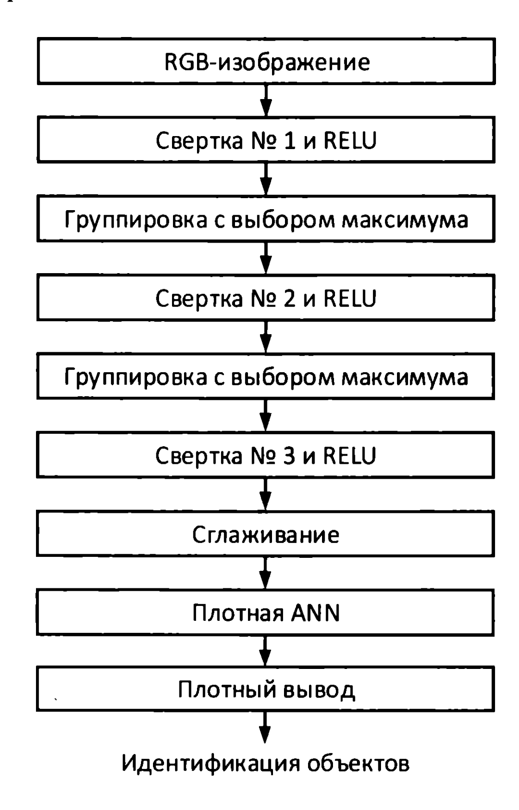


Рис. 16.1. *Уровни CNN, используемые для идентификации объектов*

В упражнениях в конце этой главы разрабатывается пример CNN, основанной на структуре, показанной на рис. 16.1, которая может идентифицировать и классифицировать объекты на цветных изображениях низкого разрешения со значительной степенью точности.

Вы можете задаться вопросом: как разработчик CNN выбирает архитектурные особенности сети, такие как типы и количество слоев и другие параметры, для конкретной прикладной задачи? Во многих случаях совсем не ясно, какой должна быть наилучшая комбинация типов слоев и других параметров сети (таких как размеры фильтра свертки) для достижения общих целей приложения на основе CNN.

Типы и размеры слоев и связанные с ними параметры, описывающие нейронную сеть, называются гиперпараметрами. **Гиперпараметр** — это высокоуровневый атрибут нейронной сети, который определяет некоторую часть ее структуры. Гиперпараметры отличаются от весовых коэффициентов (см. рис. 6.4) на нейронных связях, которые определяются автоматически в процессе обучения.

Выбор значений гиперпараметра для настройки нейронной сети с целью решения конкретной прикладной задачи можно рассматривать как процесс поиска. Разработчик нейронной сети может использовать программные средства для тестирования различных архитектур сети, состоящих из различных последовательностей разнотипных слоев нейронной сети, а также разных значений параметров, связанных с каждым слоем, таких как размеры фильтров свертки или количество нейронов в полносвязном слое. Процесс поиска включает в себя этап обучения для каждой конфигурации сети, за которым следует этап тестирования для оценки характеристик обученной сети. Проекты сетей, которые показали наилучшие результаты на этапе тестирования, сохраняются для дальнейшей оценки по мере продвижения процесса проектирования.

В следующем разделе мы рассмотрим пример использования лидара для определения местоположения транспортного средства и распознавания объектов в его окружении.

Определение местоположения с помощью лидара

Как говорилось ранее, лидары измеряют расстояние до поверхностей, отражающих часть излучаемой лазером энергии обратно в направлении лидара. Направление передаваемого лазерного луча относительно транспортного средства во время каждой серии измерений известно программному обеспечению обработки данных лидара. Расстояние до точки отражения и обратно вычисляется с помощью измерения интервала времени между моментом испускания лазерного импульса и возвращением эхосигнала на лидар, который называется временем прохождения импульса. Расстояние, пройденное импульсом до цели и обратно, равно времени прохождения импульса, умноженному на скорость света.

В результате быстрых последовательных измерений во всех направлениях вокруг транспортного средства формируется облако точек, которое содержит полный набор измерений для точек вокруг транспортного средства.

Сравнивая созданное лидаром облако точек с сохраненной картой поверхностей и сооружений вблизи местоположения транспортного средства, программное обеспечение обработки данных лидара может корректировать и подстраивать полученное в результате измерений облако точек, чтобы привести его в соответствие с данными сохраненной карты. Такой процесс позволяет лидарной системе точно опреде-

лять свое местоположение и ориентацию относительно окружающей обстановки. Этот подход называется **локализацией**.

По сравнению с системами автономного вождения, основанными на анализе видеоизображений, системы вождения на основе лидара обладают следующими преимуществами.

- Лидарные датчики непрерывно создают точные трехмерные карты окружающей обстановки. Эта информация представляет собой расстояние от датчика до каждой точки в облаке точек. Одна видеокамера создает двумерное изображение, не содержащее информации о расстоянии до какой-либо точки сцены.
- Видеокамеры могут быть ослеплены солнечным светом или другим ярким источником света. Лидарные датчики гораздо менее восприимчивы к таким помехам.
- Функция обнаружения объектов в системах на основе видеокамер может не распознавать объекты, когда воспринимаемый цвет объекта недостаточно контрастирует с окружающей средой. Лидарные системы обнаруживают объекты в своем окружении независимо от их цвета.

Лидарные системы имеют следующие ограничения.

- Они имеют существенно меньшее разрешение, чем стандартные видеокамеры.
- Как и в случае с видеокамерами, характеристики лидарных датчиков могут ухудшиться из-за погодных условий, таких как туман и сильный дождь.
- Лидарные системы локализации работают только на дорогах и в других средах, которые были точно нанесены на карту. База данных, содержащая информацию о карте, должна быть доступна транспортному средству во время его эксплуатации. Это требует постоянных вложений для сбора картографических данных о дорогах, используемых для автономного вождения. При составлении карт необходимо повторять процесс сбора данных, чтобы выявлять такие изменения, как строительство новых зданий вблизи дорог. При движении по дорогам, которые не нанесены на карту, лидарная система будет недоступна для использования.
- Лидарные системы, как правило, значительно дороже видеокамер.

Между автопроизводителями и технологическими компаниями продолжается конкуренция за разработку и внедрение систем автономного вождения на основе различных технологий наблюдения за окружающей обстановкой. С 2021 г. Tesla сосредоточила свои усилия на использовании нескольких видеокамер для предоставления информации, необходимой для выполнения функций автономного вождения без точной карты окружающей среды.

Другие компании, включая Toyota и Waymo, полагаются на использование лидара в качестве основной системы для контроля окружающей обстановки в сочетании с базой данных, содержащей трехмерные данные для локализации.

Независимо от того, использует система автономного вождения видеокамеры или лидарную систему, она должна распознавать объекты и отслеживать их поведение во времени, чтобы выполнять функции, необходимые для безопасного вождения. Это тема следующего раздела.

Отслеживание объектов

Выходные данные системы автономного вождения после этапа распознавания включают в себя список объектов и их классификацию (например, грузовик или велосипедист), полученные на основе данных видеокамеры или лидара. Для поддержки решения задачи вождения необходимо выполнить дополнительную обработку для ведения истории объектов, обнаруженных в последовательных выборках данных от датчиков, и отслеживания поведения этих объектов с течением времени.

Функции отслеживания объектов оценивают движение объектов и позволяют прогнозировать их движение в будущем. Результаты прогнозирования движения объектов используются при принятии решений о планировании траектории для поддержания безопасных дистанций до других объектов и уменьшения риска столкновений.

Отслеживание объектов в последовательных выборках данных позволяет выявлять и исправлять ошибки, такие как обнаружение ложных объектов в отдельных сценах, если присутствие предполагаемого объекта не может быть подтверждено последующими выборками.

После получения набора данных, описывающего состояние транспортного средства и всех объектов, имеющих отношение к управлению им, система автономного вождения должна принять ряд решений, которые будут рассмотрены далее.

Принятие решений

Используя информацию, полученную на этапе обработки данных, поступающих от датчиков, система автоматического вождения принимает решения о действиях, которые она будет выполнять в каждый момент для продолжения безопасного движения транспортного средства в запрошенный пункт назначения. Основные функции, которые она должна поддерживать, заключаются в удержании автомобиля в пределах полосы движения, соблюдении правил дорожного движения, предотвращении столкновений с другими объектами и планировании траектории движения. Эти темы рассматриваются в следующем разделе.

Удержание полосы движения

Задача удержания полосы движения требует, чтобы водитель транспортного средства (человек или автономная система) постоянно контролировал его положение в пределах занимаемой полосы, не допуская превышения допустимого отклонения от центра полосы движения.

Удержание полосы движения — простая задача на дороге с четкой разметкой в хороших погодных условиях. С другой стороны, система автономного вождения (или

водитель-человек) может испытывать значительные трудности с удержанием полосы движения, когда разметка полосы движения скрыта свежим снегом или грязью на поверхности дороги.

Система автономного вождения на основе лидара не должна испытывать значительных затруднений при движении автомобиля по дороге, покрытой тонким слоем снега, пока лидарные измерения окружающей обстановки продолжают предоставлять достоверную информацию. При этом система, основанная на видеокамерах, в таких условиях может столкнуться с существенными трудностями.

Соблюдение правил дорожного движения

Системы автономного вождения должны соблюдать все правила дорожного движения и нормативные требования при эксплуатации автомобиля на дорогах общего пользования. Сюда относятся базовые функции, такие как остановка перед знаками "Стоп" и надлежащее реагирование на сигналы светофора, а также другие необходимые действия, такие как предоставление права проезда другому транспортному средству, когда этого требует ситуация.

Задача вождения может усложниться, когда автономным транспортным средствам придется делить дорогу с автомобилями, управляемыми людьми. Люди выработали целый ряд моделей поведения для различных дорожных ситуаций, которые призваны облегчить вождение для всех участников. Например, автомобиль на переполненном шоссе может притормозить, чтобы освободить место для выезда на проезжую часть другого автомобиля, вливающегося в транспортный поток с примыкающей дороги.

Успешная система автономного вождения должна соответствовать всем законодательным требованиям и демонстрировать поведение, отвечающее ожиданиям водителей-людей.

Предотвращение столкновений

Системы автономного вождения должны распознавать и соответствующим образом реагировать на всю гамму неодушевленных предметов и живых существ, с которыми водители сталкиваются на дорогах ежедневно. Это касается не только других транспортных средств, пешеходов и велосипедистов — случайные предметы, такие как лестницы, автомобильные капоты, шины и ветви деревьев, также оказываются на дорогах каждый день. Различные животные, например белки, кошки, собаки, олени, крупный рогатый скот, медведи и лоси, регулярно пытаются пересечь дорогу в различных регионах.

Водители-люди обычно стараются избегать наезда на мелких животных на дороге, если это можно сделать, не создавая неприемлемо опасной ситуации. При встрече с крупным животным предотвращение столкновения может оказаться необходимым, чтобы не допустить гибели пассажиров транспортного средства. Ожидается, что системы автономного вождения должны превосходить людей-водителей во всех этих ситуациях.

Планирование траектории движения

При высокоуровневом планировании маршрута составляется последовательность связанных друг с другом участков дорог, по которым автомобиль должен проехать во время движения от начальной точки к пункту назначения. Системы планирования маршрутов на основе GPS-приемников хорошо знакомы водителям современных транспортных средств. Автономные транспортные средства используют тот же подход для планирования маршрута к пункту назначения.

Низкоуровневое планирование маршрута включает в себя все действия по управлению на пути к пункту назначения. Система автономного вождения должна постоянно оценивать окружающую обстановку и принимать решения, например, о том, когда менять полосу движения, когда безопасно въезжать на перекресток и когда отказаться от попыток повернуть налево, на улицу с оживленным движением, и вернуться к альтернативному маршруту.

Как и во всех аспектах автономного вождения, высшими целями планирования маршрута являются обеспечение безопасности пассажиров и других лиц за пределами транспортного средства, соблюдение всех правил дорожного движения и максимально быстрое перемещение к пункту назначения с соблюдением вышеуказанных требований.

Вычислительная архитектура автономного транспортного средства

На рис. 16.2 приведена схема взаимосвязи аппаратных компонентов и этапов обработки в системе автономного вождения на основе текущего состояния технологий, описанных в этой главе.

Мы представили технологии контроля обстановки, которые помогают собирать информацию о состоянии автомобиля и его окружения. Эта информация поступает в процесс *Наблюдение*, который принимает данные от датчиков, проверяет правильность работы каждого датчика и подготавливает данные для распознавания. Процесс *Распознавание* принимает необработанные данные датчиков и извлекает из них полезную информацию, такую как выявление объектов на видеоизображениях, определение их местоположения и скорости. После достижения точного понимания состояния транспортного средства и всех выявленных окружающих объектов процесс *Принятие решений* реализует навигационные функции высокого уровня, такие как выбор маршрута движения в пункт назначения, а также функции низкого уровня, такие как выбор направления движения на перекрестке. На основе принятых решений процесс *Действия* отправляет команды аппаратным блокам, выполняющим руление, регулирующим скорость транспортного средства и предоставляющим информацию другим водителям, в первую очередь посредством управления сигналами поворота.

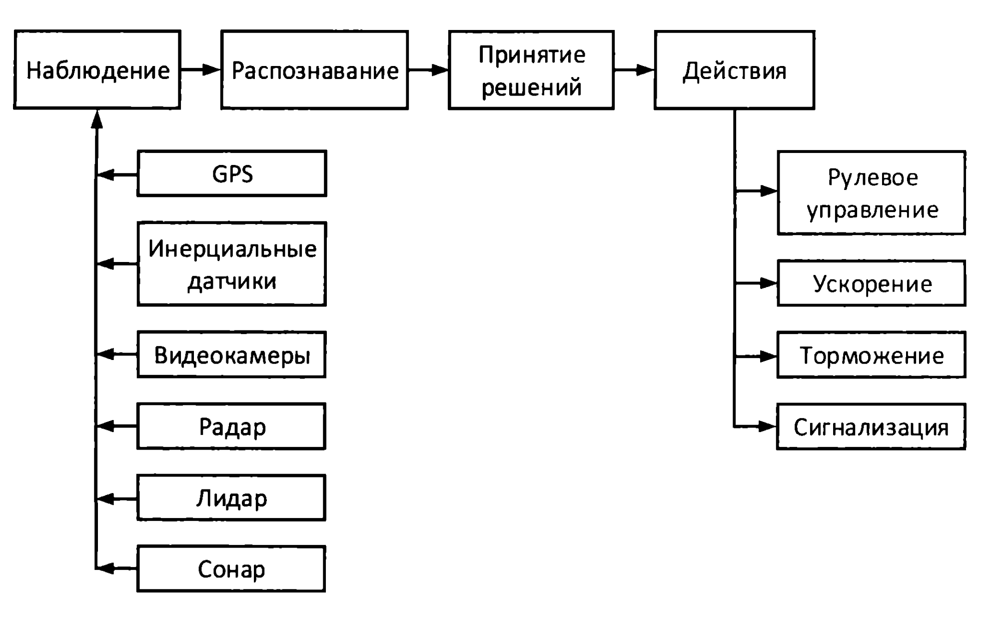


Рис. 16.2. *Компоненты и процессы автономной системы вождения*

Автопилот Tesla HW3

Для того чтобы сохранить коммерческую тайну, некоторые автомобильные и технологические компании, разрабатывающие системы автономного вождения, публикуют крайне ограниченную информацию об устройстве своих систем. В то же время компания Tesla была более откровенна в отношении информации о компьютерном оборудовании для автономного вождения, используемом в производимых ею автомобилях. Автомобили Tesla в настоящее время эксплуатируются на дорогах общего пользования с так называемыми возможностями полного автономного вождения, основанными на использовании компьютерной системы под названием **Hardware 3.0** (**HW3**).

Плата процессора Tesla HW3 содержит две полностью резервированные вычислительные системы, каждая из которых способна безопасно и самостоятельно управлять автомобилем. Обе системы работают синхронно, непрерывно сравнивая свои выходные данные. Если одна из систем выходит из строя, вторая берет управление автомобилем на себя до тех пор, пока не будет выполнен ремонт.

Компьютер HW3 создан на основе специализированной системы на кристалле (SoC), разработанной Tesla для оптимизации его рабочих характеристик при решении задач автономного вождения. В дополнение к традиционным возможностям, связанным с обработкой изображений и скалярными вычислениями, на чипе SoC размещены значительные ресурсы для вычислений с использованием нейронных сетей.

Ниже перечислены некоторые из основных особенностей компьютера Tesla HW3.

- Интегральная схема SoC изготовлена с применением 14-нанометрового процесса FinFET и содержит в общей сложности 6 млрд транзисторов. Техноло-

гия **FinFET** (Fin Field-Effect Transistor) использует вертикальную конструкцию (называемую "плавником"), отличающую ее от плоской структуры традиционной технологии КМОП. Транзисторы FinFET обеспечивают более быстрое переключение и пропускают больший ток, чем традиционные КМОП-транзисторы. Компьютер HW3 стал первым примером применения 14-нанометровой технологии FinFET в автомобилестроении.

- В качестве DRAM в компьютере HW3 используются схемы **DDR4 с пониженным энергопотреблением** (**LPDDR4**). LPDDR4 — это вариант DRAM DDR4, оптимизированный для снижения энергопотребления в таких устройствах, как смартфоны и ноутбуки. LPDDR4 имеют меньшую пропускную способность по сравнению с DDR4 и потребляют значительно меньше энергии. Аккумулятор автомобиля Tesla имеет гораздо бóльшую емкость по сравнению с аккумулятором смартфона. Тем не менее электроэнергию, потребляемую автомобильным компьютером, необходимо свести к минимуму, и использование DRAM LPDDR4 помогает в решении этой задачи.
- Каждый чип SoC содержит два ускорителя нейронных сетей, которые вместе обеспечивают скорость 72 **тераопераций в секунду** (**TOPS**). Эти процессоры используют нейросети CNN для обнаружения объектов на изображениях видеокамер.
- Каждый ускоритель нейронной сети имеет 32 Кбайт выделенной высокоскоростной памяти SRAM. Как было показано в *главе 8*, схемы памяти SRAM намного быстрее DRAM, но имеют значительно более высокую стоимость за бит с точки зрения площади чипа. Значительный объем SRAM обеспечивает повышение производительности по сравнению с другими процессорами общего назначения, которые Tesla могла бы использовать вместо своих специализированных SoC.
- Для вычислений общего назначения каждый процессор HW3 содержит три четырехъядерных 64-разрядных центральных процессора ARM Cortex A72, работающих на частоте 2,2 ГГц.

Согласно данным Tesla, благодаря архитектуре CNN, реализованной в компьютере HW3, он может обрабатывать впечатляющий объем видео высокой четкости — 2300 кадров в секунду. Столь высокий уровень производительности необходим для достижения заявленной Tesla цели — использовать датчики на основе видеокамер для обеспечения автономного вождения 5-го уровня.

Технологии автономных транспортных средств стремительно развиваются, но текущий уровень оснащаемых ими автомобилей, доступных для приобретения широкой публикой, по-прежнему значительно отстает от цели беспилотной эксплуатации. Может пройти несколько лет, прежде чем появятся автомобили, способные перевозить пассажиров по дорогам общего пользования без участия человека, постоянно наблюдающего за окружающей обстановкой через лобовое стекло.

Резюме

В этой главе были представлены возможности, которые должны быть реализованы в вычислительных архитектурах самоуправляемых автомобилей. Она началась с представления уровней автономности вождения и обсуждения требований по обеспечению безопасности самоуправляемого автомобиля и его пассажиров, а также безопасности других транспортных средств, пешеходов и стационарных объектов. Затем мы рассмотрели типы датчиков и источники информации, которые предоставляют самоуправляемым автомобилям входные данные во время движения. Также мы обсудили типы вычислений, необходимых для эффективного управления автомобилем. Глава завершилась обзором вычислительной архитектуры автомобильного компьютера Tesla HW3.

Эта глава помогла вам ознакомиться с основами вычислительной архитектуры для самоуправляемых автомобилей и с типами датчиков, используемых такими автомобилями. Теперь вы должны разбираться в типах вычислений, применяемых в самоуправляемых автомобилях, и понимать связанные с ними проблемы безопасности.

В следующей, заключительной главе мы обсудим перспективы развития компьютерных архитектур. В ней будут рассмотрены важные достижения и современные тенденции, которые привели к текущему состоянию компьютерных архитектур, а также перспективы дальнейшего развития этих тенденций для определения некоторых возможных будущих технологических направлений. Также будут рассмотрены потенциально прорывные технологии, которые могут изменить направление развития компьютерных архитектур в будущем. В заключение будут предложены определенные подходы к профессиональному совершенствованию архитектора компьютеров, которые могут помочь сформировать набор полезных для будущего навыков.

Упражнения

1. Если на вашем компьютере еще не установлен Python, перейдите по адресу **https://www.python.org/downloads/** и установите текущую версию. Убедитесь, что файлы Python включены в пути поиска, набрав `python --version` в командной строке системы. Вы должны получить ответ, подобный следующему: `Python 3.10.3`.

   Установите TensorFlow (платформу для машинного обучения с открытым исходным кодом) с помощью команды (также в системной командной строке) `pip install tensorflow`. Возможно, для успешной установки вам потребуется использовать функцию запуска от имени администратора при открытии окна командной строки. Установите Matplotlib (библиотеку для визуализации данных) с помощью команды `pip install matplotlib`.

2. Используя библиотеку TensorFlow, создайте программу, которая загружает набор данных CIFAR-10 и отображает подмножество изображений вместе с мет-

ками, связанными с каждым изображением. Этот набор данных является продуктом **Канадского института перспективных исследований** (Canadian Institute for Advanced Research, CIFAR) и содержит 60 000 изображений, каждое из которых состоит из 32 × 32 RGB-пикселов. Эти изображения были случайным образом разделены на обучающий набор из 50 000 изображений и тестовый набор из 10 000 изображений. Каждое изображение было помечено людьми как представляющее предмет одной из 10 категорий: самолет, автомобиль, птица, кошка, олень, собака, лягушка, лошадь, судно или грузовик. Для получения дополнительной информации о наборе данных CIFAR-10 см. технический отчет Алекса Крижевски (Alex Krizhevsky) по адресу **https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf**.

3. Используя библиотеку TensorFlow, создайте программу, которая строит сверточную нейросеть (CNN) на основе структуры, показанной на рис. 16.1. Примените фильтр свертки 3 × 3 в каждом сверточном слое. Используйте 32 фильтра в первом сверточном слое и 64 фильтра в двух других сверточных слоях. В скрытом слое используйте 64 нейрона. Выделите 10 выходных нейронов, отражающих отнесение изображения к одной из 10 категорий CIFAR-10.

4. Используя библиотеку TensorFlow, создайте программу, которая обучает нейросеть CNN, разработанную в *упражнении 3*, и испытайте итоговую модель с помощью тестовых изображений набора CIFAR-10. Определите долю правильно идентифицированных нейросетью тестовых изображений.

17

Квантовые вычисления и другие перспективные направления в вычислительных архитектурах

В этой главе описаны дальнейшие перспективы развития компьютерных архитектур. Мы рассмотрим важные технологические достижения и современные тенденции, которые привели к текущему состоянию компьютерных архитектур. Затем мы попробуем экстраполировать текущие тенденции и определим некоторые возможные направления развития вычислительных систем в будущем. Мы также рассмотрим некоторые потенциально прорывные технологии, которые могут существенно повлиять на развитие будущих компьютерных архитектур.

В этой главе представлены некоторые рекомендуемые подходы к профессиональному развитию архитектора компьютеров. Следуя этим рекомендациям, вы сможете поддерживать набор навыков, который останется актуальным и совместимым с будущими достижениями, какими бы они ни оказались.

Прочитав эту главу, вы познакомитесь с историей развития компьютерных архитектур, которая привела к нынешней ситуации в этой области, а также с текущими тенденциями в проектировании компьютеров и их возможным влиянием на будущие технологические направления. Вы получите представление о некоторых потенциально прорывных технологиях, которые могут существенно изменить компьютерные архитектуры будущего. Вы также изучите некоторые полезные методы для поддержания неизменно актуального набора навыков в области компьютерной архитектуры.

В этой главе будут представлены следующие темы:

- текущее развитие компьютерных архитектур;
- экстраполяция современных тенденций в будущее;

- потенциально прорывные технологии;
- формирование набора навыков с заделом на будущее.

Технические требования

Файлы для этой главы, включая ответы к упражнениям, доступны по адресу **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organization-Second-Edition**.

Текущее развитие компьютерных архитектур

В *главе 1* было представлено краткое введение в историю автоматических вычислительных устройств, от механической аналитической машины Бэббиджа до появления архитектуры *x*86, которая продолжает служить основой для большинства современных персональных компьютеров. Это развитие вычислительных технологий опиралось на ряд новаторских технологических достижений, в первую очередь на изобретение транзистора и разработку процессов изготовления интегральных схем.

За десятилетия, прошедшие с момента появления Intel 4004 в 1971 г., процессоры значительно усложнились в части количества транзисторов и других компонентов схемы, интегрированных в общий кристалл. Одновременно с ростом количества элементов схемы в расчете на чип на несколько порядков увеличилась тактовая частота устройств.

Расширение возможностей процессора и увеличение скорости выполнения инструкций привело к превращению разработки программного обеспечения в отдельную огромную отрасль мирового масштаба. На заре цифровых компьютеров программное обеспечение разрабатывалось небольшими группами высококвалифицированных специалистов в исследовательской обстановке. Сегодня мощные персональные компьютеры можно приобрести за вполне разумные деньги, а средства разработки программного обеспечения, такие как компиляторы и интерпретаторы языков программирования, находятся в широком доступе и зачастую бесплатны. Поскольку возможности процессоров значительно расширились, доступность и распространенность вычислительных ресурсов создали высокий спрос на программное обеспечение для работы на этих устройствах.

В интегральную схему современного процессора заложено гораздо больше функциональных возможностей, чем в ранние устройства, такие как процессор 6502. 6502 содержит базовый набор компонентов, необходимый для выполнения полезных вычислений: устройство управления, набор регистров, арифметико-логическое устройство и внешняя шина для доступа к инструкциям, данным и периферийным устройствам.

Самые сложные современные процессоры, предназначенные для бизнеса и домашних пользователей, включают в себя базовую функциональность, аналогичную

возможностям 6502, с существенными дополнительными функциями и расширениями:

- до 16 процессорных ядер, каждое из которых поддерживает одновременную многопоточность;
- многоуровневая кеш-память инструкций и данных;
- кеш-память для микроопераций, помогающая избежать задержки обработки, связанной с повторяющимися операциями декодирования инструкций;
- блок управления памятью, поддерживающий виртуальную память со страничной организацией;
- встроенная поддержка многоканального высокоскоростного последовательного ввода-вывода;
- встроенный графический процессор, формирующий сигналы для цифрового видеовыхода;
- поддержка запуска виртуализированных операционных систем.

Подводя итог технологическому развитию от процессора 6502 до современного процессора *x*64, можно сказать, что в отличие от одного 8-разрядного ядра 6502 современные процессоры оснащаются несколькими параллельно работающими 64-разрядными ядрами и реализуют множество дополнительных функций, разработанных специально для увеличения скорости выполнения инструкций.

В дополнение к высокой чистой вычислительной мощности процессоров современных ПК набор инструкций *x*86/*x*64 позволяет выполнять широкий спектр операций — от простых до чрезвычайно сложных. С другой стороны, в современных RISC-процессорах, таких как ARM и RISC-V, реализованы намеренно сокращенные наборы инструкций с целью разбиения сложных операций на последовательности более простых шагов, каждый из которых выполняется с очень высокой скоростью при работе в контексте более широкого набора регистров.

Высокоуровневые конфигурации вычислительных архитектур, пожалуй, не претерпели кардинальных изменений со времен процессора 6502. С каждым расширением набора инструкций архитектуры процессора или внедрением дополнительной технологии кеширования эти изменения постепенно расширяли функциональность, доступную разработчикам программного обеспечения, или увеличивали скорость выполнения алгоритмов. Переход к поддержке нескольких ядер и многопоточности в пределах одного ядра обеспечил возможность одновременного выполнения нескольких независимых потоков вместо использования принципа разделения по времени на одном ядре.

Постепенный характер этого развития по большей части был преднамеренным, чтобы избежать внесения в процессорную архитектуру изменений, которые могли воспрепятствовать обратной совместимости с множеством уже разработанных операционных систем и прикладного программного обеспечения. Конечный результат — серия поколений процессоров, которые постепенно становятся все быстрее и производительнее, но не реализуют прорывные технологии прошлого.

В следующем разделе мы попытаемся экстраполировать текущее состояние поколения высокопроизводительных вычислительных систем, рассмотренных в *главе 13*, чтобы предсказать подвижки в области компьютерных архитектур, которые могут произойти в течение следующих одного-двух десятилетий.

Экстраполяция современных тенденций в будущее

Возможности процессорных технологий текущего поколения начинают упираться в некоторые существенные физические ограничения, которые, как мы можем ожидать, будут сдерживать темпы роста производительности в будущем. Эти ограничения, конечно, не приведут к резкой остановке прогресса в области повышения плотности схем и тактовой частоты. Скорее, улучшение характеристик будущих поколений процессоров будет происходить по направлениям, отличающимся от традиционных моделей роста возможностей полупроводниковых устройств. Для того чтобы более внимательно изучить ожидания роста производительности процессоров в будущем, мы начнем с того, что вернемся к закону Мура и рассмотрим его применимость к будущему полупроводниковых технологий.

Закон Мура — новый взгляд

Пересмотренная версия закона Мура, опубликованная Гордоном Муром в 1975 г., предсказывала, что количество компонентов интегральных схем в расчете на устройство будет удваиваться примерно каждые два года. Этот закон демонстрировал замечательную точность прогнозирования в течение нескольких десятилетий, но по состоянию на 2015 г., по данным Intel, темпы роста замедлились до удвоения примерно каждые два с половиной года. Это указывает на то, что темпы роста плотности интегральных схем начали замедляться, но этот рост, безусловно, не прекратился и, как ожидается, не прекратится и в обозримом будущем.

Технологии интегральных схем продолжат совершенствоваться, что обеспечит появление более высокопроизводительных устройств с большей плотностью компонентов в течение многих лет. Однако мы можем ожидать, что скорость роста плотности схем со временем начнет снижаться из-за физических ограничений, связанных с созданием компонентов размером порядка единиц нанометров.

Замедление темпов роста плотности схем не означает, что эта тенденция близка к завершению. По состоянию на 2022 г., современные технологии массового производства интегральных схем основаны на элементах с размерами всего около 5 нм. В настоящее время ведется работа по разработке следующего поколения схемотехнических решений с размерами элементов 4 нм. Будущие поколения с размерами элементов 3 и даже 2 нм находятся на стадии планирования.

Через какое-то время такое увеличение плотности схем, вероятно, будет реализовано, но технологический прогресс сопряжен с ростом затрат и техническими проблемами, которые приводят к задержкам в развертывании новых технологий на

производственных линиях. Самые передовые технологии производства интегральных схем настолько дороги в разработке и сложны во внедрении, что лишь несколько крупных полупроводниковых компаний обладают финансовыми ресурсами и техническим опытом для внедрения таких процессов.

Учитывая продолжающееся снижение темпов роста плотности схем, производители полупроводников начали уделять особое внимание альтернативным методам, позволяющим более плотно размещать компоненты на чипе. Традиционно интегральные схемы рассматривались в основном как двумерные структуры, формируемые послойно следующим образом:

- последовательное нанесение различных материалов в ходе операций маскирования для создания легированных областей транзисторов, а также других компонентов схемы, таких как конденсаторы и диоды;
- нанесение токопроводящих металлизированных дорожек на элементы схемы в виде дополнительных слоев.

Связь между элементами схемы при двумерной компоновке устройства предусматривает электрическое взаимодействие между объектами, расположенными на поверхности чипа на некотором расстоянии друг от друга. Чип имеет небольшие размеры, поэтому время, необходимое электрическому сигналу для прохождения между объектами, часто незначительно по сравнению с другими ограничениями.

Вы можете задаться вопросом: а нельзя ли разместить элементы интегральной схемы иным способом, отличным от их распределения по плоской поверхности? Действительно, элементы можно укладывать на кристалле интегральной схемы друг на друга. Этот подход к проектированию мы рассмотрим в следующем разделе.

Третье измерение

Разработав методы размещения элементов друг на друге на одном кристалле интегральной схемы, производители полупроводников сделали шаг к расширению закона Мура. Одной из первых целей для многоуровневых конфигураций интегральных схем стала вездесущая пара из *n*-канального и *p*-канального МОП-транзисторов, используемая в схемах КМОП.

Intel опубликовала данные об успехах, достигнутых ее исследователями в области многоуровневых пар КМОП-транзисторов, в начале 2020 г. Компания не только продемонстрировала способность многоуровневого размещения устройств на кремниевом кристалле, но и показала, как использовать различные технологии изготовления на каждом уровне для достижения максимальных рабочих характеристик пары транзисторов. Кремниевые *n*-канальные транзисторы обладают хорошими рабочими характеристиками, но *p*-канальные транзисторы, построенные на кремнии, имеют относительно меньшую скорость переключения. *p*-Канальные транзисторы, реализованные с германиевым транзисторным каналом вместо кремниевого, обеспечивают повышенную скорость переключения, улучшая производительность КМОП-пары.

В демонстрации интеграции устройств Intel со смешанными технологиями кремниевые *n*-канальные транзисторы были сформированы на базовом кремниевом кристалле с расположенными поверх них германиевыми *p*-канальными устройствами. Если этот метод можно масштабировать для поддержки производства интегральных схем, он обещает дальнейшее увеличение плотности устройств и повышение тактовой частоты.

Другой повышающий плотность подход заключается в объединении нескольких отдельно сконструированных кристаллов интегральных схем в вертикальный пакет с соединениями между слоями для питания и передачи данных. Этот подход можно представить, как припаивание кристаллов интегральных схем друг поверх друга.

Отдельно изготовленные интегральные схемы, объединяемые в одном пакете, называют **чиплетами**. Чиплеты можно разложить рядом друг с другом на кремниевой подложке или уложить друг на друга — в зависимости от требований к устройству. Такой подход позволяет сконструировать каждый чиплет сложного устройства с помощью наиболее подходящей для него технологии. Например, один способ изготовления может быть оптимальным для главного процессора, в то время как другой процесс может лучше всего подходить для чипсета памяти, интегрированного с процессором. Интегрированный радиоинтерфейс сотовой связи в том же пакете можно сконструировать с применением еще одного процесса.

Использование вертикального измерения при конструировании отдельных интегральных схем и сложных устройств, состоящих из нескольких чиплетов в одном пакете, обеспечивает более высокий уровень интеграции систем на кристалле (SoC) и более высокую общую производительность. Поскольку совершенствование и внедрение этих технологий на производственных линиях продолжается, мы можем ожидать, что увеличение сложности и функциональности схем, предсказанное законом Мура, продолжится и в предстоящие годы, хотя, вероятно, и с меньшими темпами.

Следующая тенденция, которую мы рассмотрим, — продолжающийся рост использования узкоспециализированных процессорных устройств вместо процессоров общего назначения.

Распространение специализированных устройств

В предыдущих главах мы рассмотрели несколько специализированных вычислительных технологий, нацеленных на такие области, как цифровая обработка сигналов, создание трехмерных графических изображений, майнинг биткоинов и обработка данных с помощью нейронных сетей. Безусловно, все вычисления, необходимые для решения этих задач, могут производиться обычными процессорами общего назначения. Важным отличием вычислений, осуществляемых специализированными устройствами, является повышенная скорость выполнения, причем пропускная способность иногда в сотни и даже тысячи раз может превысить уровень, доступный для обычного процессора.

Растущее значение машинного обучения и автономных технологий будет и дальше стимулировать инновации в компьютерных архитектурах, лежащих в основе будущих цифровых систем. По мере того как автомобили и другие сложные системы приобретают автономные функции, которые либо дополняют, либо заменяют функции, традиционно выполняемые операторами-людьми, базовые вычислительные архитектуры продолжат развиваться, чтобы обеспечить более высокий уровень производительности для решения конкретных задач при минимальном энергопотреблении.

Специализированные процессоры будут использовать преимущества достижений, рассмотренных ранее в этой главе, что поможет оптимизировать конструкции устройств для конкретных областей применения. Тенденция к росту специализации процессоров сохранится и в ближайшие годы, по всей видимости, ускорится.

Обсуждение в этом разделе было сосредоточено на продолжении текущих тенденций в предстоящие годы. В следующем разделе мы рассмотрим возможность появления технологических инноваций, которые существенно изменят путь развития и позволят перейти от непрерывной череды постепенных улучшений компьютерной архитектуры к чему-то совершенно иному.

Потенциально прорывные технологии

До сих пор в этой главе основное внимание уделялось текущим тенденциям и возможным последствиям их дальнейшего развития в будущем. На примере ситуации с изобретением транзистора мы увидели, что всегда возможно появление новой технологии, которая радикально меняет предыдущий опыт и ведет будущее вычислительной техники в новом направлении.

В этом разделе мы попытаемся определить некоторые потенциальные источники таких технологических достижений в ближайшие годы.

Квантовая физика

В своей аналитической машине Чарльз Бэббидж пытался довести возможности чисто механических вычислительных устройств до ранее недостижимого уровня. Его попытка, несмотря на амбициозность замысла, в конечном счете оказалась неудачной. Разработке применимых на практике автоматизированных вычислительных устройств пришлось подождать до тех пор, пока появление технологии вакуумных ламп не обеспечило подходящую основу для реализации сложной цифровой логики.

Позже изобретение транзистора вывело вычислительные технологии на путь наращивания возможностей и сложности, который в конечном счете привел нас к тому состоянию вычислительной техники, которым мы наслаждаемся сегодня. С момента появления процессора Intel 4004 прогресс в возможностях вычислений принял форму постепенных улучшений того, что по сути является все той же технологией, положенной в основу кремниевых транзисторов.

Работа транзисторов основана на свойствах полупроводниковых материалов, таких как кремний, и применении этих свойств для реализации цифровых коммутационных схем. Цифровые схемы, построенные на полупроводниках, обычно выполняют операции с использованием двоичного формата данных. Эти устройства предназначены для получения надежно воспроизводимых результатов при вводе одних и тех же данных при последующем выполнении одной и той же последовательности инструкций.

В качестве альтернативы этому подходу по всему миру проводятся многочисленные исследовательские работы, изучающие возможность использования аспектов квантовой физики в вычислительной технике. **Квантовая физика** описывает поведение материи на уровне отдельных атомов и субатомных частиц.

Поведение частиц на субатомном уровне существенным и удивительным образом отличается от знакомого поведения макрообъектов, с которыми мы взаимодействуем каждый день по законам **классической физики**. Законы квантовой физики были открыты и описаны в теориях, зародившихся в середине XIX в.

Квантовая физика строго определяется набором теорий, которые продемонстрировали замечательную предсказательную способность. Например, Вольфганг Паули постулировал существование частицы нейтрино в рамках квантовой физики в 1930 г. Нейтрино — это крошечные субатомные частицы, которые практически не взаимодействуют с другими частицами, что делает их обнаружение чрезвычайно трудным. Существование нейтрино не было доказано научными экспериментами до 1950-х годов.

Несколько других типов субатомных частиц были предсказаны теорией и в конечном счете обнаружены экспериментально. Квантовая физика, включая странное поведение, проявляющееся в субатомной сфере, предлагает многообещающее новое направление для компьютерных архитектур будущего.

Изменение физических параметров, связанных с макрообъектами, например скорости движущегося автомобиля по мере его ускорения или замедления, кажется непрерывным. С другой стороны, электроны внутри атома могут существовать только на определенных, дискретных энергетических уровнях. Энергетический уровень электрона в атоме примерно соответствует скорости движения частицы по орбите вокруг центрального тела в классической физике.

Электрон в атоме не может существовать в какой-либо точке между двумя энергетическими уровнями. Он всегда находится строго на том или ином энергетическом уровне. Дискретный характер уровней энергии привел к введению термина **"квант"** для описания таких явлений.

Спинтроника

Помимо энергетического уровня в атоме электроны обладают свойством, аналогичным вращению объекта в классической физике. Спин элементарной атомной частицы представляет собой особый тип углового момента, концептуально аналогичный моменту импульса вращающегося баскетбольного мяча, балансирующего

на кончике пальца. Как и в случае с энергетическим уровнем, это спиновое состояние квантуется.

Исследователи продемонстрировали способность менять и измерять спиновое поведение электронов способом, который может оказаться подходящим для практических цифровых коммутационных схем. Использование спина электрона в качестве компонента цифровой коммутационной схемы называют **спинтроникой** — это слово получено путем объединения терминов *spin* (вращение) и *electronics* (электроника). Эта технология использует квантовое спиновое состояние электронов для хранения информации аналогично состоянию заряда конденсаторов в традиционной электронике.

Однако есть некоторые существенные отличия спинового поведения электронов от вращения баскетбольных мячей. Электроны на самом деле не вращаются — скорее, их спиновое поведение подчиняется математическим законам углового момента в квантованной форме. Баскетбольный мяч можно заставить вращаться с произвольно выбранной скоростью, в то время как электроны могут демонстрировать вращение только на одном дискретном, квантованном уровне. Спин элементарной частицы определяется ее типом, и электроны всегда имеют спин (квантовое число), равный 1/2.

Вращение баскетбольного мяча можно полностью охарактеризовать сочетанием скорости его вращения и направления оси, вокруг которой происходит вращение. Раскрученный мяч, балансирующий на кончике пальца, вращается вокруг вертикальной оси. Вращательное движение мяча можно полностью описать вектором, направленным вдоль оси вращения (в данном случае вверх), с величиной, равной скорости вращения мяча.

Электроны всегда имеют одно и то же значение спина, определяющее длину вектора углового момента, поэтому единственный способ отличить спин одного электрона от другого — по направлению вектора спина. Были разработаны практические устройства, которые могут обеспечить выравнивание векторов спина электрона в двух различных ориентациях — *вверх* и *вниз*. Сочетание значения спина 1/2 и его ориентации образует **спиновое квантовое число**.

Спин электрона формирует крошечное магнитное поле. Материалы, в которых спины большинства электронов выровнены по направлению, создают магнитное поле с той же ориентацией, что и у большинства выровненных электронов. Эффект этого выравнивания электронов наблюдается в обычных магнитных материалах, таких как магниты для холодильника.

Магнитное поле, создаваемое спином электрона, нельзя объяснить классической физикой. Этот тип магнетизма является исключительно эффектом квантовой физики.

Коммутационное устройство, называемое **спиновым клапаном**, можно построить на основе канала внутри интегральной схемы, на каждой стороне которого имеется магнитный слой. Магнитные слои выполняют функцию затворов. Если затворы имеют одинаковую полярность спина (вверх или вниз), через устройство может протекать ток, состоящий из электронов, поляризованных по спину. Если затворы имеют противоположную полярность, ток блокируется. Спиновый клапан можно

включать и выключать, меняя полярность одного из магнитов путем подачи на него тока с противоположным направлением спина.

Переключение направления спина электрона может осуществляться намного быстрее и потреблять гораздо меньше энергии, чем процесс зарядки и разрядки конденсаторов, который лежит в основе функционирования современных цифровых устройств, использующих технологию КМОП.

Эта ключевая особенность позволяет рассматривать потенциал спинтроники как технологии, которая со временем может дополнить или заменить КМОП-схемы в высокопроизводительных цифровых устройствах.

В настоящее время спинтроника — это область активных исследований. Коммерческое освоение и производство цифровых устройств, которые превосходят современные КМОП-процессоры, вряд ли начнутся в течение ближайших нескольких лет, если эта технология вообще окажется жизнеспособной.

Спинтроника полагается на законы квантовой физики для выполнения цифровой коммутации. Квантовые вычисления, предмет следующего раздела, напрямую используют квантово-механические явления для реализации аналоговой и цифровой обработки.

Квантовые вычисления

Квантовые вычисления обещают резкое повышение скорости выполнения определенных классов задач. **Квантовые вычисления** выполняют обработку с помощью квантово-механических явлений и могут использовать для решения задач аналоговые или цифровые подходы.

В цифровых квантовых вычислениях для выполнения вычислительных операций применяют квантовые логические вентили, которые основаны на схемах, называемых **квантовыми битами** или **кубитами**. Кубиты в некотором роде аналогичны битам в традиционных цифровых компьютерах, но между ними есть существенные различия. Традиционные биты могут принимать только состояния 0 и 1. Кубит может находиться в квантовом состоянии 0 или 1; однако он также может существовать в суперпозиции состояний 0 и 1 — это означает, что у него есть некоторая вероятность появления либо 0, либо 1 при наблюдении. Принцип **квантовой суперпозиции** утверждает, что результатом сложения любых двух допустимых квантовых состояний будет допустимое квантовое состояние.

Всякий раз, когда считывается значение кубита, всегда возвращается либо 0, либо 1. Это связано со "схлопыванием" суперпозиции квантовых состояний в одно состояние. Если до считывания кубит содержал квантовое значение, соответствующее двоичному значению 0 или 1, результат операции чтения будет равен двоичному значению. Если, с другой стороны, кубит содержал суперпозицию состояний, значение, возвращаемое операцией чтения, будет вероятностной функцией суперпозиции состояний.

Другими словами, вероятность получения 0 или 1 в результате чтения кубита зависит от характеристик его квантового состояния. Значение, возвращаемое операцией

чтения, будет непредсказуемым. Причина этой непредсказуемости не просто в недостатке знаний; в квантовой физике частица просто не имеет определенного состояния, пока не будет проведено измерение. Это одна из нелогичных и, откровенно говоря, трудно вообразимых особенностей квантовой физики.

Состояние кубита, которое *близко* к двоичному значению 1, будет иметь более высокую вероятность возвращения значения 1 при чтении, чем то, которое ближе к двоичному значению 0. Выполнение операции чтения для нескольких кубитов, которые изначально находились в одинаковых квантовых состояниях, не всегда будет давать один и тот же результат из-за вероятностного характера операции чтения.

Схемы на основе кубитов могут демонстрировать и использовать свойства квантовой запутанности — центрального принципа квантовой физики. **Квантовая запутанность** возникает, когда несколько частиц связаны таким образом, что измерение одной частицы влияет на измерение связанных с ней частиц. Наиболее удивительный аспект этой связи заключается в том, что она остается в силе даже тогда, когда частицы находятся на большом расстоянии друг от друга. Эффект запутанности, по-видимому, распространяется мгновенно, не ограничиваясь скоростью света. Такое поведение может показаться научной фантастикой, однако оно было подтверждено экспериментально и даже использовалось в коммуникационных технологиях космического аппарата NASA **Lunar Atmosphere Dust and Environment Explorer** (**LADEE**), который находился на орбите вокруг Луны с 2013 по 2014 г.

Квантовые компьютеры способны использовать запутанность при обработке информации. Если вы проработаете примеры в конце этой главы, у вас будет возможность создать программу для квантового компьютера, демонстрирующую эффекты квантовой запутанности, и вы сможете запустить эту программу на реальном квантовом компьютере.

Несколько непредсказуемый характер результатов, возвращаемых при чтении кубита, по-видимому, выступает против использования этой технологии в качестве основы для цифровой вычислительной системы. Эта частичная непредсказуемость — одна из причин, по которой квантовые компьютеры считаются полезными только для определенных классов задач. Большинству клиентов не понравится, если банк будет использовать компьютер, который каждый раз вычисляет разные остатки на счетах из-за квантовой неопределенности.

В следующих разделах описаны две ключевые категории областей применения, которые в настоящее время рассматриваются для квантовых компьютеров.

Квантовый взлом кода

Квантовый взлом кода использует методы цифровых квантовых вычислений для взлома современных криптографических кодов. Многие применяемые сегодня криптографические алгоритмы основаны на предположении, что с вычислительной точки зрения определение множителей большого числа (содержащего, возможно, сотни десятичных цифр), являющегося произведением двух больших простых чисел, — невыполнимая задача. Нельзя ожидать, что разложение на множители тако-

го числа на современных компьютерах или даже на суперкомпьютере или тысячах процессоров, работающих параллельно в облачной среде, даст правильный результат за разумный период времени.

Алгоритм Шора, разработанный Питером Шором (Peter Shor) в 1994 г., описывает действия, которые должен выполнить квантовый компьютер, чтобы определить простые множители данного числа. Квантовый компьютер, использующий алгоритм Шора, потенциально способен разложить очень большое число на множители гораздо быстрее, чем обычные компьютеры, тем самым делая современные криптографические системы, основанные на криптографии с открытым ключом, уязвимыми для таких атак. На сегодняшний день из-за ограниченного количества кубитов в квантовых компьютерах квантовые вычисления продемонстрировали только способность определять множители относительно небольших чисел, таких как 21, но потенциальную угрозу признают организации и правительства, которым требуется высокий уровень безопасности связи. В будущем могут появиться системы квантовых вычислений, способные взламывать коды, которые мы используем сегодня для защиты веб-сайтов и онлайн-банкинга.

Однако нет особых причин беспокоиться о надежности вашего банковского счета в связи с угрозой квантовых атак. В стадии разработки уже находится множество алгоритмов шифрования с открытым ключом, устойчивых к взлому с применением квантовых вычислений. В совокупности эти алгоритмы называются **постквантовой криптографией**. В случае если квантовая угроза современным методам криптографии станет реальной, можно ожидать масштабного перехода к квантово-устойчивым криптографическим алгоритмам.

Адиабатические квантовые вычисления

Адиабатические квантовые вычисления — это подход к аналоговым квантовым вычислениям, который обещает эффективное решение широкого круга практических задач оптимизации. Представьте, что вы находитесь в холмистой местности на прямоугольном участке, огражденном забором. Вам нужно найти самую низкую точку в пределах огороженной области. В этом сценарии стоит сильный туман, и вы не можете видеть окружающую местность. Единственная подсказка, которая у вас есть, — это наклон поверхности под вашими ногами. Вы можете идти вниз по склону, но при достижении ровного участка вы не можете быть уверены в том, где находитесь, — в местной котловине или действительно нашли самую низкую точку во всей огражденной области.

Это пример простой задачи двумерной оптимизации. Цель состоит в том, чтобы найти в заданной области координаты x и y точки с наименьшей высотой, называемой **глобальным минимумом**, не блуждая и не застревая во впадине на большей высоте, называемой **локальным минимумом**.

Для того чтобы найти самую низкую точку в холмистой местности, не требуется задействовать нечто столь замысловатое, как квантовые вычисления, но многие реальные задачи оптимизации имеют значительно большее количество входных перемен-

ных, например от 20 до 30, которые необходимо менять при поиске глобального минимума. Вычислительная мощность, необходимая для решения таких задач, выходит за рамки возможностей даже самых быстрых современных суперкомпьютеров.

Подход квантовых вычислений к решению таких проблем начинается с создания конфигурации кубитов, содержащей суперпозицию всех возможных решений проблемы, и последующего медленного уменьшения эффекта суперпозиции.

Ограничивая состояние конфигурации квантовой схемы во время этого процесса, можно гарантировать, что решение, которое останется после устранения суперпозиции и определения дискретных значений 0 или 1 для всех кубитов, является глобальным минимумом.

Термин **"адиабатический"** в названии этого метода ссылается на аналогию между процессом исключения суперпозиции и термодинамической системой, которая при работе не теряет и не набирает тепло.

Адиабатическая квантовая оптимизация — это область интенсивных исследований. Еще предстоит выяснить, какой уровень возможностей эта технология способна привнести в решение сложных задач оптимизации.

Будущее квантовых вычислений

Термин **"квантовое превосходство"** описывает переходную точку, в которой квантовые вычисления превосходят возможности традиционных цифровых вычислений в конкретной проблемной области. Среди исследователей ведутся оживленные дебаты о том, было ли уже достигнуто квантовое превосходство какой-либо из крупных организаций, разрабатывающих технологии квантовых вычислений, когда эта точка может быть достигнута и может ли такой переход вообще когда-либо произойти.

Ряд существенных препятствий стоит на пути широкого внедрения квантовых вычислений способом, аналогичным повсеместному использованию вычислительных устройств на основе КМОП по всему миру сегодня. Вот некоторые из наиболее актуальных проблем, требующих решения:

- увеличение количества кубитов в компьютере для поддержки решения крупных и сложных задач;
- реализация возможности инициализировать кубиты произвольными значениями;
- предоставление механизмов для надежного считывания состояния кубитов;
- признание, что компоненты, необходимые для квантовых компьютеров, сложны в производстве и очень дороги;
- устранение эффектов квантовой декогеренции.

Термин **"квантовая декогеренция"** относится к потере фазовой когерентности в квантовой системе. Для того чтобы квантовый компьютер функционировал должным образом, внутри системы должна поддерживаться фазовая когерентность. Квантовая декогеренция возникает в результате вмешательства внешних факторов

во внутреннюю работу квантовой системы или в результате помех, генерируемых внутри системы. Квантовая система, которая остается совершенно изолированной, может поддерживать фазовую когерентность неограниченно долго. Нарушение работы системы, например, путем считывания ее состояния нарушает когерентность и может привести к декогеренции. Процесс контроля и корректировки эффектов декогеренции называют **коррекцией квантовых ошибок**.

Эффективное управление декогеренцией — одна из самых больших проблем в квантовых вычислениях.

Конструкции современных квантовых компьютеров основаны на экзотических материалах, таких как гелий-3, который нарабатывается в ядерных реакторах, и для них требуются сверхпроводящие кабели с нулевым сопротивлением. Во время работы системы квантовых вычислений должны охлаждаться до температур, близких к абсолютному нулю. Современные квантовые компьютеры в основном представляют собой лабораторные системы, и для их создания и эксплуатации требуется специально выделенная группа экспертов. Эта ситуация в чем-то аналогична ранним дням компьютеров на базе вакуумных ламп. Одно из главных отличий от времен вакуумных ламп заключается в том, что сегодня у нас есть Интернет, который предоставляет обычным пользователям доступ к возможностям квантовых вычислений.

Современные системы квантовых вычислений содержат не более одной-двух сотен кубитов и доступны по большей части коммерческим, академическим и правительственным организациям, которые финансируют их разработку. Однако существуют некоторые уникальные возможности получения доступа к реальным квантовым компьютерам для студентов и обычных людей.

Одним из примеров является IBM Quantum: **https://www.ibm.com/quantum-computing/**. С помощью этих бесплатных ресурсов компания IBM предоставляет набор инструментов, включая среду разработки квантовых алгоритмов под названием **Qisket**, доступную по адресу **https://www.qiskit.org/**. Используя инструменты Qisket, разработчики могут научиться создавать квантовые алгоритмы и даже отправлять программы для выполнения в пакетном режиме на реальный квантовый компьютер. Упражнения в конце этой главы описывают действия, которые следует предпринять, чтобы начать работу в этой области.

Квантовые вычисления демонстрируют значительные перспективы для решения определенных категорий задач, хотя широкая коммерциализация этой технологии, судя по всему, произойдет не ранее, чем через несколько лет.

Следующая технология, которую мы рассмотрим, — это углеродные нанотрубки, которые потенциально могут хотя бы частично отодвинуть цифровую обработку от мира кремния.

Углеродные нанотрубки

Полевой транзистор на углеродных нанотрубках (carbon nanotube field-effect transistor, CNTFET) — это транзистор, который в качестве канала затвора использует одну углеродную нанотрубку либо массив таких нанотрубок вместо кремниевого

канала традиционного полевого МОП-транзистора. Углеродная нанотрубка — это составленная из атомов углерода трубчатая структура диаметром приблизительно 1 нм.

Углеродные нанотрубки являются исключительно хорошими электрическими проводниками, обладают высокой прочностью на растяжение и очень хорошо проводят тепло. Углеродная нанотрубка может выдерживать плотность тока более чем в 1000 раз бóльшую, чем такие металлы, как медь. В отличие от металлов, электрический ток может распространяться только вдоль оси нанотрубки.

Ниже перечислены преимущества транзисторов CNTFET по сравнению с полевыми МОП-транзисторами.

- Более высокий управляющий ток.
- Существенно меньшая рассеиваемая мощность.
- Устойчивость к высоким температурам.
- Превосходное рассеивание тепла, обеспечивающее высокую плотность размещения устройств.
- Рабочие характеристики *n*- и *p*-канальных устройств CNTFET полностью совпадают. С другой стороны, в КМОП-устройствах характеристики *n*- и *p*-канальных транзисторов могут существенно различаться. Это ограничивает общие рабочие характеристики схемы возможностями устройства с наихудшими рабочими характеристиками.

Как и в случае с другими новыми технологиями, обсуждаемыми в этой главе, технология CNTFET сталкивается с некоторыми существенными препятствиями на пути коммерциализации и широкого применения.

- Изготовление CNTFET-транзисторов чрезвычайно сложно из-за необходимости укладывать и перемещать трубки нанометрового размера.
- Производство нанотрубок, необходимых для CNTFET-транзисторов, также очень сложный процесс. Нанотрубки можно рассматривать как листы углеродной ткани, которые необходимо свернуть в трубки вдоль определенной оси, чтобы получить материал с желаемыми полупроводниковыми свойствами.
- Углеродные нанотрубки быстро разлагаются под воздействием кислорода. Технологии изготовления должны учитывать это ограничение, чтобы обеспечить долговечность и надежность получаемой схемы.

Учитывая сложность массового производства CNTFET-транзисторов, скорее всего, пройдет несколько лет, прежде чем в коммерческих устройствах начнут широко применяться транзисторы на основе углеродных нанотрубок, если это вообще произойдет.

В предыдущих разделах были рассмотрены некоторые передовые технологии (спинтроника, квантовые вычисления и транзисторы на основе углеродных нанотрубок) как перспективные области, которые могут внести существенный вклад в будущее вычислительной техники. На момент написания текста ни одна из этих технологий не использовалась в широких масштабах, но исследования показали

многообещающие результаты, и многие правительственные, университетские и коммерческие лаборатории усердно работают над развитием этих технологий и поиском путей их применения в вычислительных устройствах будущего.

Помимо таких широко обсуждаемых технологий, которые, судя по всему, развиваются по предсказуемому хотя бы отчасти пути, всегда существует вероятность того, что какая-либо организация или частное лицо объявит о непредвиденном технологическом прорыве. Это может произойти в любой момент, и такое событие может перевернуть общепринятые представления о предполагаемом пути развития компьютерных технологий в будущем. Только время покажет.

В контексте неопределенности будущего вычислительных архитектур профессионалу в этой области целесообразно разработать стратегию, которая обеспечит постоянную актуальность его знаний независимо от того, по какому пути будут развиваться будущие технологии. В следующем разделе представлены некоторые рекомендации насчет того, как поддерживать свою компетентность в области технологических достижений.

Формирование набора навыков с заделом на будущее

Учитывая технологические изменения, которые положили начало эре цифровых вычислений на основе транзисторов, и возможность подобных событий в будущем, для профессионалов в области архитектуры компьютеров важно идти в ногу с текущими достижениями и развивать интуицию относительно вероятных направлений, в которых эти технологии будут развиваться. В данном разделе представлены некоторые практические рекомендации, которые помогут вам быть в курсе последних достижений в области компьютерных технологий.

Непрерывное обучение

Специалисты по компьютерной архитектуре должны принять идею о том, что технологии продолжают быстро развиваться, и профессионалы должны постоянно прилагать значительные усилия для отслеживания достижений и учета новых разработок в своей повседневной работе и решениях по планированию профессионального роста.

Предусмотрительный профессионал полагается на широкий спектр источников информации, чтобы наблюдать за разработками перспективных технологий и оценивать их влияние на карьерные цели. Некоторые источники информации, такие как традиционные новостные сводки, можно просмотреть достаточно быстро, усвоив полезные сведения. Другие источники, к примеру научная литература и веб-сайты, курируемые экспертами в конкретных технологиях, требуют времени для изучения сложной технической информации. Более продвинутые темы, такие как квантовые вычисления, могут потребовать углубленного изучения только для

того, чтобы понять основы и оценить возможные варианты применения соответствующей технологии.

Даже при четком понимании конкретной технологии может быть сложно или даже невозможно точно предсказать ее влияние на отрасль и способы ее интеграции в архитектуры вычислительных систем, используемых коммерческими предприятиями, государственными учреждениями и широкой публикой.

Практичный и простой в реализации подход к сбору информации состоит в том, чтобы выделить набор надежных источников общих и технических новостей и постоянно контролировать публикуемую в них информацию. Основные СМИ, включая телевизионные новости, газеты, журналы и веб-сайты, часто публикуют информацию о многообещающих технологических разработках и влиянии цифровых устройств на общество во всем мире. Помимо обсуждения чисто технических аспектов вычислительных систем (до некоторой степени), эти источники предоставляют сведения о влиянии вычислительных технологий на общество, например опасения по поводу их использования для государственной и корпоративной слежки или распространения дезинформации.

Технические веб-сайты, контролируемые исследовательскими организациями, отдельными экспертами в области технологий и энтузиастами-любителями, предлагают огромное количество информации, связанной с достижениями в области компьютерной архитектуры. Как и в случае со всей информацией в Интернете, рекомендуется учитывать надежность источника всякий раз, когда вы сталкиваетесь с неожиданной информацией. Ведется множество оживленных дискуссий об эффективности отдельных технологий, находящихся на ранних стадиях развития, однако есть также некоторые люди, которые выражают несогласие с опубликованной информацией просто ради спора. В конечном счете вам решать, насколько можно доверять любым мнениям, представленным на веб-странице.

Ситуация в области источников технологических новостей постоянно меняется, и у каждого человека свои предпочтения. Тем не менее в следующем списке приведены некоторые достаточно надежные источники новостей о компьютерных технологиях, указанные без особого порядка.

- **https://techcrunch.com/**. TechCrunch сообщает о коммерческих новостях в области высоких технологий.
- **https://www.wired.com/**. Wired — это ежемесячный журнал и веб-сайт, посвященный тому, как новые технологии влияют на культуру, экономику и политику.
- **https://arstechnica.com/**. Издание Ars Technica, основанное в 1998 г., публикует информацию, предназначенную для профессионалов в области информационных технологий.
- **https://www.tomshardware.com/**. Tom's Hardware предлагает новости, статьи, сравнение цен и обзоры компьютерного оборудования и высокотехнологичных устройств.

- **https://www.engadget.com/**. Технологический блог Engadget, созданный в 2004 г., освещает вопросы, входящие в сферу игр, технологий и развлечений.
- **https://gizmodo.com/**. Gizmodo уделяет особое внимание разработкам, технологиям и научной фантастике. Лозунг сайта — "Мы пришли из будущего".
- **https://thenextweb.com/**. Проект TNW был запущен в 2006 г. с целью поделиться с общественностью пониманием и представлениями о мире технологий.

Этот список, хотя и далеко не полный, может послужить отправной точкой для сбора информации о текущем состоянии и ближайшем будущем вычислительных технологий и их применении.

Информация из Интернета, если подходить к ней с достаточно скептической точки зрения, может предоставлять актуальные и точные сведения о состоянии достижений в области компьютерной архитектуры. Однако информация, полученная таким образом, не дает образования в том строгом смысле, который характерен для формального обучения, и не является публичной декларацией того, что вы усвоили эту информацию и способны применять ее в профессиональном контексте.

Диплом о высшем образовании, о котором пойдет речь в следующем разделе, дает основательную подготовку по дисциплине и обычно принимается потенциальными работодателями и клиентами как свидетельство приобретения профессиональных навыков.

Высшее образование

Если прошло несколько лет с тех пор, как вы в последний раз посещали учебное заведение, или если вы начали свою карьеру без высшего образования, возможно, пришло время рассмотреть вопрос о его получении. Если даже мысль о том, чтобы предпринять подобное, кажется вам невозможной из-за работы или семейных обстоятельств, следует учесть, что многие аккредитованные учебные заведения, предлагающие отличные программы обучения в областях, непосредственно связанных с архитектурой компьютеров, предоставляют возможность полноценного обучения через Интернет. Онлайн-обучение в сочетании с очной сдачей экзаменов дает возможность получить степень бакалавра или магистра по техническим дисциплинам в ряде самых авторитетных университетов мира.

В случае работников с образованием, проработавших несколько лет, — технологии и аналитические методы, которые они изучали в своем учебном заведении, могли в какой-то степени устареть. Для того чтобы восстановить актуальность знаний и поддержать компетентность в области передовых технологий, связанных с проектированием и производством современных компьютерных систем, лучшим подходом может быть возвращение в аудиторию для более подробного ознакомления с техническими достижениями, которые произошли за прошедшие годы.

Если вы не готовы к участию в программе очного обучения, многие учебные заведения предлагают онлайн-курсы, предусматривающие выдачу сертификата в такой предметной области, как разработка или производство компьютерного оборудования. Несмотря на меньшую авторитетность по сравнению со степенью бакалавра

или магистра, прохождение программы технологической сертификации все же демонстрирует определенный уровень образования и знаний в предметной области.

Для прохождения курсов потребуются некоторые расходы на плату за обучение и учебники независимо от того, проводится обучение очно или через Интернет. Некоторые работодатели готовы предоставить частичное или полное финансирование участия своих сотрудников в аккредитованных программах обучения. Это финансирование может сопровождаться обязательством студента продолжать работу у работодателя в течение определенного периода времени после завершения обучения. Студенты должны внимательно изучить и понять любые обязательства, которые они могут взять на себя, если обстоятельства потребуют от них ухода из учебного заведения или увольнения из компании-работодателя.

Информацию о программах онлайн-обучения для получения высшего образования или сертификатов, которые соответствуют вашим потребностям, можно найти на множестве веб-сайтов. Вот некоторые примеры для США:

- **https://www.usnews.com/education/online-education**: по этому адресу U.S. News & World Report публикует ежегодные рейтинги аккредитованных колледжей, включая, в том числе, онлайн-программы;
- **https://www.onlineu.com/**: на веб-сайте OnlineU представлены отзывы студентов, проходящих онлайн-обучение в сотнях колледжей.

Стараясь не наскучить вам своими предупреждениями, напомню, что вам следует внимательно изучить любую информацию, почерпнутую из Интернета относительно онлайн-обучения. Убедитесь, что любое рассматриваемое учебное заведение имеет надлежащую аккредитацию, а присуждаемые им степени принимают и ценят работодатели.

Те, у кого есть необходимые ресурсы или поддержка со стороны работодателя, могут даже рассмотреть возможность очной формы обучения на время прохождения программы. Работодатели, которые платят за обучение своих сотрудников, как правило, ожидают, что студент согласится принять на себя некоторые обязательства перед организацией после прохождения учебной программы. Такой подход может обеспечить самый быстрый путь к получению диплома и, во многих случаях, открывает возможности участия в исследованиях самых передовых компьютерных технологий, находящихся в стадии разработки.

Высшее образование в уважаемом учебном заведении и в соответствующей области обучения является "золотым стандартом", востребованным работодателями и признаваемым коллегами, но быть в курсе последних исследований можно также посредством участия в конференциях и чтения научной литературы. Такие возможности обучения представлены в следующем разделе.

Конференции и литература

Для профессионалов, заинтересованных в том, чтобы быть в курсе передовых исследований в области технологий, связанных с компьютерными архитектурами будущего, возможно, нет лучшего способа, чем узнать о последних разработках от

самих исследователей. По всему миру регулярно проводятся конференции по любым темам, связанным с передовыми вычислительными технологиями, которые вы только можете себе представить. Например, список проводимых по всему миру конференций на тему квантового поведения, включая многие мероприятия, посвященные аспектам квантовых вычислений, доступен по адресу **http://quantum.info/conf/index.html**.

Как и в случае с другой информацией из Интернета, полезно относиться к любой незнакомой конференции с некоторой долей скептицизма, пока вы ее тщательно не проверите. К сожалению, существует явление, известное как **мусорные конференции**, когда своекорыстные отдельные лица или организации проводят конференции с целью получения дохода, а не для обмена научными знаниями. Убедитесь, что любая конференция, на которую вы регистрируетесь и которую собираетесь посетить, контролируется авторитетной организацией и содержит презентации законных исследователей в предметных областях, имеющих отношение к конференции.

Существует большое разнообразие научной литературы, связанной с текущими достижениями в технологиях по компьютерной архитектуре. Профессиональные организации, такие как IEEE, публикуют многочисленные научные журналы, посвященные передовым достижениям современных исследований. Эти журналы предназначены для прямого общения между исследователями, поэтому уровень технических знаний, ожидаемый от читателей, довольно высок. Если у вас есть необходимая подготовка и вы готовы анализировать подробности в статьях, публикуемых в научных журналах, можете читать эти публикации, чтобы поддерживать уровень своих знаний наравне с учеными и инженерами, разрабатывающими вычислительные технологии следующего поколения.

Резюме

Давайте кратко рассмотрим темы, которые мы обсуждали и о которых узнали в главах этой книги.

- В *главе 1 "Введение в архитектуру компьютеров"* мы начали с самого раннего проекта автоматической вычислительной машины — аналитической машины Бэббиджа — и проследили историю развития цифровых компьютеров от самых ранних вычислительных устройств на базе вакуумных ламп до первых поколений процессоров. Мы также ознакомились с архитектурой раннего, но все еще распространенного микропроцессора — 6502.
- В *главе 2 "Цифровая логика"* мы изучили основы транзисторных технологий, цифровой логики, регистров и последовательностной логики. Мы также обсудили использование языков описания аппаратных средств при разработке сложных цифровых устройств.
- В *главе 3 "Элементы процессора"* были рассмотрены основные компоненты процессоров, включая устройство управления, арифметико-логическое устройство и набор регистров. В этой главе были представлены концепции, связанные с набором инструкций процессора, включая подробную информацию

о режимах адресации, категориях инструкций, обработке прерываний и операциях ввода-вывода процессора 6502.

- В *главе 4 "Компоненты компьютерных систем"* был представлен полевой МОП-транзистор и описано его использование в схемах динамической памяти DRAM. Кроме того, в этой главе рассматривались подсистемы обработки и связи современных компьютеров, включая подсистему ввода-вывода, графические дисплеи, сетевой интерфейс и интерфейсы для клавиатуры и мыши.
- В *главе 5 "Аппаратно-программный интерфейс"* мы узнали о внутреннем устройстве драйверов и о том, как встроенное программное обеспечение BIOS оригинального ПК было заменено UEFI в современных компьютерах. В этой главе были описаны процесс загрузки и концепции, связанные с процессами и потоками в современных операционных системах.
- *Глава 6 "Специализированные вычисления"* представила уникальные возможности вычислений в реальном времени, цифровой обработки сигналов и обработки в графическом процессоре. Были описаны примеры специализированных вычислительных архитектур, основанных на этих возможностях обработки, включая облачные серверы, настольные компьютеры для бизнеса и высокопроизводительные игровые компьютеры.
- *Глава 7 "Архитектура процессора и памяти"* была посвящена уникальным особенностям фон-неймановской, гарвардской и модифицированной гарвардской архитектур. В этой главе было описано различие между физической и виртуальной памятью и представлена архитектура виртуальной памяти со страничной организацией, включая функции блока управления памятью.
- В *главе 8 "Методы повышения производительности"* мы узнали о множестве методов, используемых в современных процессорах для повышения скорости выполнения инструкций. Рассматривались такие темы, как кеш-память, конвейерная обработка инструкций, суперскалярная обработка, одновременная многопоточность и обработка по принципу "одна инструкция, множество данных" (SIMD).
- В *главе 9 "Специализированные расширения процессоров"* был рассмотрен ряд дополнительных возможностей процессора, включая привилегированные режимы выполнения, арифметику с плавающей запятой, управление питанием и управление безопасностью системы.
- В *главе 10 "Современные архитектуры и наборы инструкций процессоров"* мы углубились в детали архитектур и наборов инструкций наиболее распространенных 32-разрядных и 64-разрядных процессоров. Для каждой процессорной архитектуры — *x*86, *x*64, 32-разрядных и 64-разрядных процессоров ARM — в главе были представлены набор регистров, режимы адресации и категории команд, а также короткая, но функциональная программа на языке ассемблера.
- В *главе 11 "Архитектура и набор инструкций RISC-V"* мы подробно рассмотрели особенности архитектуры RISC-V. В этой главе была представлена базовая 32-разрядная архитектура, включая набор регистров, набор инструкций

и стандартные расширения к набору инструкций. Дополнительные темы включали рассмотрение 64-разрядной версии архитектуры и стандартных конфигураций, доступных в качестве процессоров RISC-V, выпускаемых серийно. Здесь было приведено несколько простых программ на языке ассемблера RISC-V и даны рекомендации по реализации процессора RISC-V на основе недорогой программируемой логической интегральной схемы (ПЛИС).

- В *главе 12 "Виртуализация процессоров"* были представлены концепции, связанные с виртуализацией процессоров, включая проблемы, которые должны преодолевать средства виртуализации. Были обсуждены методы, используемые для внедрения виртуализации в современных семействах процессоров, включая *x*86, ARM и RISC-V. Было описано несколько популярных инструментов виртуализации, и представлены подходы к виртуализации, используемые в средах облачных вычислений.
- В *главе 13 "Специализированные компьютерные архитектуры"* мы ознакомились с примерами некоторых конкретных компьютерных архитектур, включая смартфоны, персональные компьютеры, облачные вычислительные среды уровня центра обработки данных и нейронные сети. Были рассмотрены уникальные требования к обработке, связанные с каждой из этих архитектур, и обсуждался подбор аппаратных средств процессоров для достижения оптимального компромисса между стоимостью, производительностью и энергопотреблением в каждом конкретном случае.
- В *главе 14 "Архитектуры для обеспечения кибербезопасности и конфиденциальности вычислений"* были представлены вычислительные архитектуры, подходящие для областей применения, где требуются исключительные гарантии безопасности. Столь высокий уровень защиты необходим в критически важных областях, включая системы национальной безопасности и обработку финансовых транзакций. Эти системы должны быть устойчивы к широкому спектру угроз кибербезопасности, в том числе к проникновению вредоносного кода, атакам через скрытые каналы и путем физического доступа к аппаратным средствам компьютеров.
- *Глава 15 "Архитектуры блокчейна и майнинга биткоинов"* начинается с краткого введения в концепции, связанные с блокчейном, открытым, криптографически защищенным реестром, содержащим последовательность транзакций. Далее последовал обзор процесса майнинга биткоинов, который добавляет транзакции в блокчейн Bitcoin и вознаграждает тех, кто выполняет эту задачу, оплатой в биткоинах. Для майнинга биткоинов требуется высокопроизводительное вычислительное оборудование, которое было представлено с точки зрения компьютерной архитектуры для майнинга биткоинов текущего поколения.
- В *главе 16 "Архитектуры для самоуправляемых автомобилей"* были представлены возможности, которые должны быть реализованы в вычислительных архитектурах самоуправляемых автомобилей. Глава началась с обсуждения требований по обеспечению безопасности самоуправляемого автомобиля

и его пассажиров, а также других транспортных средств, пешеходов и стационарных объектов. Затем мы рассмотрели типы датчиков и источники информации, которые предоставляют самоуправляемым автомобилям входные данные во время движения. После этого было приведено описание типов вычислений, необходимых для эффективного управления автомобилем. Глава завершилась обзором примера архитектуры компьютера для самоуправляемого автомобиля.

В этой главе мы попытались получить некоторое представление о будущем компьютерных архитектур. Мы рассмотрели основные достижения и современные тенденции, которые привели к текущей ситуации в области проектирования компьютеров, и попытались экстраполировать их, чтобы определить направления, в которых архитектуры вычислительных систем могут развиваться в будущем. Мы также обсудили некоторые потенциально прорывные технологии, способные существенно повлиять на пути развития будущих компьютерных архитектур. Вам по силам заглянуть в будущее — выполнив упражнения в конце этой главы, вы сможете разработать алгоритм квантовых вычислений и запустить его на реальном квантовом компьютере бесплатно!

В этой главе также были рассмотрены некоторые рекомендуемые подходы к профессиональному развитию архитектора компьютеров, способствующие формированию набора навыков, который останется актуальным и совместимым с будущими достижениями, какими бы они ни оказались.

Прочитав эту главу и эту книгу, вы ознакомились с историей развития компьютерных архитектур с самых первых дней до их текущего состояния, а также с текущими тенденциями в разработке компьютерных архитектур, которые могут указать будущие технологические направления.

Вы также получили представление о некоторых потенциально прорывных технологиях, которые могут существенно изменить архитектуру компьютеров в ближайшем будущем. Наконец, в этой главе были предложены некоторые полезные методы для поддержания неизменно актуального набора навыков в области компьютерной архитектуры.

Итак, мы подошли к концу книги. Я надеюсь, что вы получили удовольствие от ее прочтения и выполнения упражнений — так же, как я, когда писал ее и выполнял эти упражнения сам.

Упражнения

1. Установите платформу разработки программного обеспечения для квантовых процессоров Qiskit, следуя указаниям по адресу: **https://qiskit.org/documentation/getting_started.html**. Эти указания предполагают установку набора инструментов для обработки данных и машинного обучения Anaconda (**https://www.anaconda.com/**). После установки пакета Anaconda создайте виртуальную среду Conda с именем `qisketenv`, где будет проходить ваша работа над квантовым кодом, и установите Qisket в этой среде с помощью команды `pip`

install qiskit. Убедитесь, что вы установили дополнительные зависимости визуализации с помощью команды pip install qiskit-terra[visualization].

2. Создайте бесплатную учетную запись IBM Quantum по адресу **https://quantum-computing.ibm.com/**. Найдите свой токен для IBM Quantum Services API на веб-сайте **https://quantum-computing.ibm.com/account** и установите его в свою локальную среду, используя указания по адресу **https://qiskit.org/documentation/stable/0.24/install.html**.
3. Рассмотрите пример квантовой программы на веб-сайте **https://qiskit.org/documentation/tutorials/circuits/1_getting_started_with_qiskit.html**. В этом примере создается квантовая схема, содержащая три кубита, которая реализует состояние **Гринбергера — Хорна — Цайлингера** (ГХЦ — Greenberger — Horne — Zeilinger, GHZ). Состояние ГХЦ проявляет ключевые свойства квантовой запутанности. Выполните код в среде моделирования на своем компьютере.
4. Выполните код из *упражнения 3* на квантовом компьютере IBM.

Приложение

Ответы к упражнениям

Глава 1. Введение в архитектуру компьютеров

Упражнение 1

Используя свой любимый язык программирования, разработайте модель одноразрядного десятичного сумматора, который работает так же, как сумматор аналитической машины Бэббиджа. Сначала запросите у пользователя две цифры в диапазоне 0–9: слагаемое и аккумулятор. Отобразите слагаемое, аккумулятор и признак переноса, который изначально равен нулю. Выполните серию циклов следующим образом:

1) если слагаемое равно 0, выведите на экран значения слагаемого, аккумулятора и признака переноса, после чего завершите программу;
2) уменьшите слагаемое на 1 и увеличьте значение аккумулятора на 1;
3) если значение аккумулятора увеличивается с 9 до 0, установите признак переноса;
4) вернитесь к *шагу 1*;

Протестируйте свой код на примере следующих операций сложения: 0 + 0, 0 + 1, 1 + 0, 1 + 2, 5 + 5, 9 + 1 и 9 + 9.

Ответ. Файл Python Ex__1_single_digit_adder.py содержит код сумматора:

```
#!/usr/bin/env python

"""Ex__1_single_digit_adder.py: ответ на упражнение 1 главы 1."""

import sys

# Выполним один шаг операции сложения аналитической машины.
# a и b - слагаемые, c - признак переноса.
def increment_adder(a, b, c):
    a = a - 1           # Уменьшим слагаемое на 1
    b = (b + 1) % 10    # Увеличим аккумулятор на 1,
                        # при необходимости выполним переход к 0
```

```
    if b == 0:          # Если аккумулятор равен 0, увеличим признак переноса на 1
        c = c + 1

    return a, b, c

# Сложим две десятичные цифры, переданные в командной строке.
# Сумма возвращается в виде digit2 и признака переноса (0 или 1).
def add_digits(digit1, digit2):
    carry = 0

    while digit1 > 0:
        [digit1, digit2, carry] = increment_adder(
        digit1, digit2, carry)

    return digit2, carry
```

Файл Ex__1_test_single_digit_adder.py содержит код тестов:

```
#!/usr/bin/env python

"""Ex__1_test_single_digit_adder.py: тесты для ответа на
упражнение 1 главы 1."""

import unittest
import Ex__1_single_digit_adder

class TestSingleDigitAdder(unittest.TestCase):
    def test_1(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        0, 0), (0, 0))

    def test_2(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        0, 1), (1, 0))

    def test_3(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        1, 0), (1, 0))
```

```
    def test_4(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        1, 2), (3, 0))

    def test_5(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        5, 5), (0, 1))

    def test_6(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        9, 1), (0, 1))

    def test_7(self):
        self.assertEqual(Ex__1_single_digit_adder.add_digits(
        9, 9), (8, 1))

if __name__ == '__main__':
    unittest.main()
```

Для того чтобы выполнить тесты, при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__1_test_single_digit_adder.py
```

Результат выполнения тестов:

```
C:\>python Ex__1_test_single_digit_adder.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s
OK
```

Упражнение 2

Создайте для слагаемого, аккумулятора и признаков переноса массивы из 40 десятичных цифр каждый. Запросите у пользователя два целых десятичных числа длиной до 40 цифр каждое. Выполните сложение чисел, цифра за цифрой, используя циклы, описанные в *упражнении 1*, и соберите выходные данные в виде признаков переноса из позиции каждой цифры в массиве признаков переноса. После завершения циклов вставьте переносы и, при необходимости, распространите их по цифрам, чтобы завершить операцию сложения. Выводите результаты на экран после

каждого цикла и по завершении. Проведите тест с теми же суммами, что и в *упражнении 1*, а также проверьте суммы 99 + 1, 999999 + 1, 49 + 50 и 50 + 50.

Ответ. Файл Python `Ex__2_40_digit_adder.py` содержит код сумматора:

```
#!/usr/bin/env python

"""Ex__2_40_digit_adder.py: ответ на упражнение 2 главы 1."""

import sys
import Ex__1_single_digit_adder

# Сложим два десятичных числа длиной до 40 цифр и возвратим сумму.
# Входные и выходные числовые значения представлены в виде строк.
def add_40_digits(str1, str2):
    max_digits = 40

    # Преобразуем str1 в десятичное значение длиной 40 цифр
    num1 = [0]*max_digits
    for i, c in enumerate(reversed(str1)):
        num1[i] = int(c) - int('0')

    # Преобразуем str2 в десятичное значение длиной 40 цифр
    num2 = [0]*max_digits
    for i, c in enumerate(reversed(str2)):
        num2[i] = int(c) - int('0')

    # Просуммируем цифры в каждой позиции и запишем
    # признак переноса для каждой позиции
    sum = [0]*max_digits
    carry = [0]*max_digits
    for i in range(max_digits):
        (sum[i], carry[i]) = Ex__1_single_digit_adder. \
        add_digits(num1[i], num2[i])

    # Распределим значения признаков переноса по цифрам
    for i in range(max_digits-1):
        if (carry[i] == 1):
            sum[i+1] = (sum[i+1] + 1) % 10
```

```
            if (sum[i+1] == 0):
                carry[i+1] = 1

    # Преобразуем результат в строку с удаленными
    # начальными нулями
    sum.reverse()
    sum_str = "".join(map(str, sum))
    sum_str = sum_str.lstrip('0') or '0'
    return sum_str
```

Файл `Ex__2_test_40_digit_adder.py` содержит код тестов:

```
#!/usr/bin/env python

"""Ex__2_test_40_digit_adder.py: тесты для ответа на упражнение 2 главы 1."""

import unittest
import Ex__2_40_digit_adder

class Test40DigitAdder(unittest.TestCase):
    def test_1(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "0", "0"), "0")

    def test_2(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "0", "1"), "1")

    def test_3(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "1", "0"), "1")

    def test_4(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "1", "2"), "3")

    def test_5(self):

        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "5", "5"), "10")
```

```
    def test_6(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "9", "1"), "10")

    def test_7(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "9", "9"), "18")

    def test_8(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "99", "1"), "100")

    def test_9(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "999999", "1"), "1000000")

    def test_10(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "49", "50"), "99")

    def test_11(self):
        self.assertEqual(Ex__2_40_digit_adder.add_40_digits(
        "50", "50"), "100")

if __name__ == '__main__':
    unittest.main()
```

Для того чтобы выполнить тесты, при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__2_test_40_digit_adder.py
```

Результат выполнения тестов:

```
C:\>python Ex__2_test_40_digit_adder.py
...........
----------------------------------------------------------------------
Ran 11 tests in 0.002s
OK
```

Упражнение 3

Измените программы *упражнений 1* и *2* так, чтобы реализовать вычитание 40-значных десятичных значений. Выполняйте заимствование по мере необходимости. Проверьте программу на примере разностей 0 – 0, 1 – 0, 1000000 – 1 и 0 – 1. Каков результат операции 0 – 1?

Ответ. Файл Python `Ex__3_single_digit_subtractor.py` содержит код одноразрядного вычитателя:

```
#!/usr/bin/env python

"""Ex__3_single_digit_subtractor.py: ответ на упражнение 3 главы 1
(одноразрядный вычитатель)."""

import sys

# Выполним один шаг операции вычитания аналитической машины.
# a и b - уменьшаемое и вычитаемое (a - b),
# c - признак переноса: 0 = с заимствованием, 1 = без заимствования
def decrement_subtractor(a, b, c):
    a = (a - 1) % 10 # Уменьшим левый операнд или на 1, или до 9 в случае переноса
    b = b - 1        # Уменьшим аккумулятор на 1

    if a == 9:       # Если аккумулятор достиг 9, уменьшим признак переноса на 1
        c = c - 1

    return a, b, c

# Выполним вычитание двух десятичных цифр. Разность возвращается в виде digit1,
# а признак переноса равен 0 (при заимствовании) или 1 (без заимствования).
def subtract_digits(digit1, digit2):
    carry = 1

    while digit2 > 0:

        [digit1, digit2, carry] = decrement_subtractor(
        digit1, digit2, carry)

    return digit1, carry
```

Файл Ex__3_test_single_digit_subtractor.py содержит код тестов для одноразрядного вычитателя:

```
#!/usr/bin/env python

"""Ex__3_test_single_digit_subtractor.py: тесты для ответа на
упражнение 3 главы 1 (тесты для одноразрядного
вычитателя)."""

import unittest
import Ex__3_single_digit_subtractor

class TestSingleDigitSubtractor(unittest.TestCase):
    def test_1(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(0, 0), (0, 1))

    def test_2(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(0, 1), (9, 0))

    def test_3(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(1, 0), (1, 1))

    def test_4(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(1, 2), (9, 0))

    def test_5(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(5, 5), (0, 1))

    def test_6(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(9, 1), (8, 1))

    def test_7(self):
        self.assertEqual(Ex__3_single_digit_subtractor.
        subtract_digits(9, 9), (0, 1))
```

```
if __name__ == '__main__':
    unittest.main()
```

Файл Python `Ex__3_40_digit_subtractor.py` содержит код 40-разрядного вычитателя:

```
#!/usr/bin/env python

"""Ex__3_40_digit_subtractor.py: ответ на упражнение 3 главы 1."""

import sys
import Ex__3_single_digit_subtractor

# Вычтем одно десятичное число длиной до 40 цифр из другого
# и возвратим результат. Входные и выходные числовые значения
# представлены в виде строк.
def subtract_40_digits(str1, str2):
    max_digits = 40

    # Преобразуем str1 в десятичное значение длиной 40 цифр.
    num1 = [0]*max_digits
    for i, c in enumerate(reversed(str1)):
        num1[i] = int(c) - int('0')

    # Преобразуем str2 в десятичное значение длиной 40 цифр.
    num2 = [0]*max_digits
    for i, c in enumerate(reversed(str2)):
        num2[i] = int(c) - int('0')

    # Вычтем цифры в каждой позиции и запишем
    # признак переноса для каждой позиции.

    diff = [0]*max_digits
    carry = [0]*max_digits
    for i in range(max_digits):
        (diff[i], carry[i]) = Ex__3_single_digit_subtractor. \
        subtract_digits(num1[i], num2[i])

    # Распределим значения признаков переноса по цифрам.
    for i in range(max_digits-1):
```

```
        if (carry[i] == 0):
            diff[i+1] = (diff[i+1] - 1) % 10
            if (diff[i+1] == 9):
                carry[i+1] = 0

    # Преобразуем результат в строку с удаленными
    # начальными нулями
    diff.reverse()
    diff_str = "".join(map(str, diff))
    diff_str = diff_str.lstrip('0') or '0'
    return diff_str
```

Файл `Ex__3_test_40_digit_subtractor.py` содержит код тестов для 40-разрядного вычитателя:

```
#!/usr/bin/env python

"""Ex__3_test_40_digit_subtractor.py: тесты для ответа на упражнение 3 главы 1."""

import unittest
import Ex__3_40_digit_subtractor

class Test40DigitSubtractor(unittest.TestCase):
    def test_1(self):
        self.assertEqual(Ex__3_40_digit_subtractor.
        subtract_40_digits("0", "0"), "0")

    def test_2(self):
        self.assertEqual(Ex__3_40_digit_subtractor.
        subtract_40_digits("1", "0"), "1")

    def test_3(self):
        self.assertEqual(Ex__3_40_digit_subtractor.
        subtract_40_digits("1000000", "1"), "999999")

    def test_4(self):
        self.assertEqual(Ex__3_40_digit_subtractor.
        subtract_40_digits("0", "1"),
        "9999999999999999999999999999999999999999")

if __name__ == '__main__':
    unittest.main()
```

Для того чтобы выполнить тесты, при условии, что Python установлен и находится по известному системе пути, выполните следующие команды:

```
python Ex__3_test_single_digit_subtractor.py
python Ex__3_test_40_digit_subtractor.py
```

Это результат выполнения тестов `Ex__3_test_single_digit_subtractor.py`:

```
C:\>python Ex__3_test_single_digit_subtractor.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s OK
```

Это результат выполнения тестов `Ex__3_test_40_digit_subtractor.py`:

```
C:\>python Ex__3_test_40_digit_subtractor.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s OK
```

Результат для 0 – 1 равен 9 с признаком переноса, равным 0.

Упражнение 4

Язык ассемблера процессора 6502 ссылается на данные в ячейках памяти, используя значение операнда, содержащее адрес (без символа #, который указывает на непосредственное значение).

Например, инструкция `LDA $00` загружает в регистр A байт, находящийся в памяти по адресу $00. `STA $01` сохраняет байт из регистра A по адресу $01. Адреса могут иметь любое значение в диапазоне от 0 до $FFFF при условии, что по указанному адресу имеется физическая память и этот адрес не используется для каких-либо других целей. С помощью предпочтительного эмулятора процессора 6502 напишите код на ассемблере 6502 для сохранения 16-битного значения по адресам $00–$01, сохраните второе значение по адресам $02–$03, затем сложите эти два значения и сохраните результат по адресам $04–$05. Обеспечьте распространение переносов между двумя байтами. Игнорируйте любой перенос из 16-битного результата. Проверьте код на примерах $0000 + $0001, $00FF + $0001 и $1234 + $5678.

Ответ. Файл на ассемблере 6502 `Ex__4_16_bit_addition.asm` содержит код сложения 16-битных значений:

```
; Ex__4_16_bit_addition.asm
; Попробуйте запустить этот код по адресу
; https://skilldrick.github.io/easy6502/
```

```
; Задайте значения для сложения.
; Удалите соответствующие точки с запятой, чтобы выбрать байты для добавления:
; ($0000 + $0001) или ($00FF + $0001) или ($1234 + $5678)

LDA #$00
;LDA #$FF
;LDA #$34
STA $00

LDA #$00
;LDA #$00
;LDA #$12
STA $01

LDA #$01
;LDA #$01
;LDA #$78
STA $02

LDA #$00
;LDA #$00
;LDA #$56
STA $03

; Сложим два 16-разрядных значения
CLC
LDA $00
ADC $02
STA $04

LDA $01
ADC $03
STA $05
```

Попробуйте запустить этот код по адресу **https://skilldrick.github.io/easy6502/**.

Упражнение 5

Напишите на ассемблере 6502 код для вычитания двух 16-разрядных значений способом, аналогичным показанному в *упражнении 4*. Проверьте код на примерах

$0001 – $0000, $0001 – $0001, $0100 – $00FF и $0000 – $0001. Каков результат операции $0000 – $0001?

Ответ. Файл на ассемблере 6502 `Ex__5_16_bit_subtraction.asm` содержит код вычитания 16-битных значений:

```
; Ex__5_16_bit_subtraction.asm
; Попробуйте запустить этот код по адресу
; https://skilldrick.github.io/easy6502/

; Задайте значения для вычитания.
; Удалите соответствующие точки с запятой, чтобы выбрать байты для
; вычитания:
; ($0001 - $0000) или ($0001 - $0001), или ($0001 - $00FF), или
; ($0000 - $0001)

LDA #$01
;LDA #$01
;LDA #$01
;LDA #$00
STA $00

LDA #$00
;LDA #$00
;LDA #$00
;LDA #$00
STA $01

LDA #$00
;LDA #$01
;LDA #$FF
;LDA #$01
STA $02

LDA #$00
;LDA #$00
;LDA #$00
```

```
;LDA #$00
STA $03

; Вычтем два 16-битных значения
SEC
LDA $00
SBC $02
STA $04

LDA $01
SBC $03
STA $05
```

Попробуйте запустить этот код по адресу **https://skilldrick.github.io/easy6502/**.

Результат для $0000 – $0001 равен $FFFF.

Упражнение 6

Напишите на ассемблере 6502 код для сохранения двух 32-разрядных целых чисел по адресам $00–$03 и $04–$07, а затем сложите их, сохранив результаты по адресам $08–$0B. Используйте циклическую конструкцию, включающую метку и команду ветвления, для перебора байтов двух складываемых значений. Найдите в Интернете подробную информацию об инструкциях уменьшения на 1 и ветвления для процессора 6502, а также об использовании меток в языке ассемблера. Подсказка: в этом приложении хорошо работает реализованный в процессоре 6502 режим адресации с индексированием нулевой страницы.

Ответ. Файл на ассемблере 6502 `Ex__6_32_bit_addition.asm` содержит код сложения 32-битных значений:

```
; Ex__6_32_bit_addition.asm
; Попробуйте запустить этот код по адресу
; https://skilldrick.github.io/easy6502/

; Задайте значения для сложения.
; Удалите соответствующие точки с запятой, чтобы выбрать байты для
; сложения:
; ($00000001 + $00000001) или ($0000FFFF + $00000001), или
; ($FFFFFFFE + $00000001), или ($FFFFFFFF + $00000001)
```

```
LDA #$01
;LDA #$FF
;LDA #$FE
;LDA #$FF
STA $00

LDA #$00
;LDA #$FF
;LDA #$FF
;LDA #$FF
STA $01

LDA #$00
;LDA #$00
;LDA #$FF
;LDA #$FF
STA $02

LDA #$00
;LDA #$00
;LDA #$FF
;LDA #$FF
STA $03

LDA #$01
STA $04

LDA #$00
STA $05
STA $06
STA $07

; Сложим два 32-битных значения, используя
; режим абсолютной индексной адресации
LDX #$00
LDY #$04
CLC
```

```
ADD_LOOP:
LDA $00, X
ADC $04, X
STA $08, X
INX
DEY
BNE ADD_LOOP
```

Попробуйте запустить этот код по адресу **https://skilldrick.github.io/easy6502/**.

Глава 2. Цифровая логика

Упражнение 1

Измените схему, представленную на рис. 2.5, чтобы превратить вентиль И в вентиль И-НЕ. Подсказка: добавлять или удалять компоненты не требуется.

Ответ. Переместите резистор R_2 и ток вывода выходного сигнала так, как показано на рис. П1.

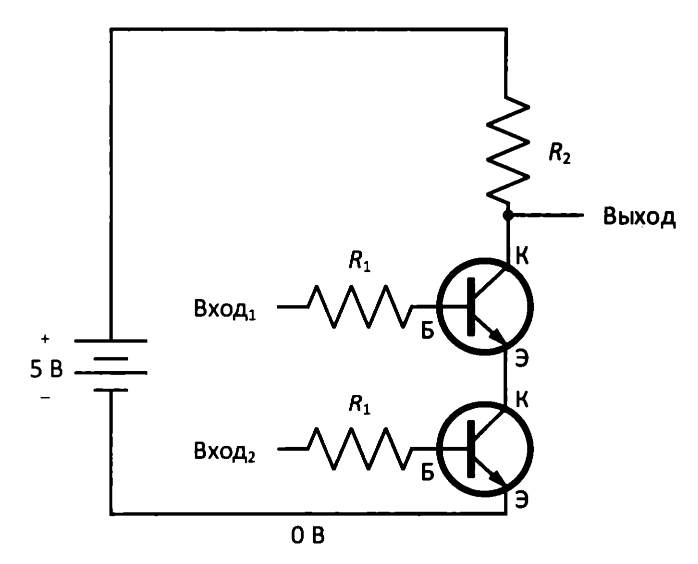


Рис. П1. *Схема вентиля И-НЕ*

Упражнение 2

Создайте реализацию схемы вентиля ИЛИ, изменив схему, представленную на рис. 2.5. По мере необходимости можно добавлять провода, транзисторы и резисторы.

Ответ. Схема вентиля ИЛИ выглядит так, как представлено на рис. П2.

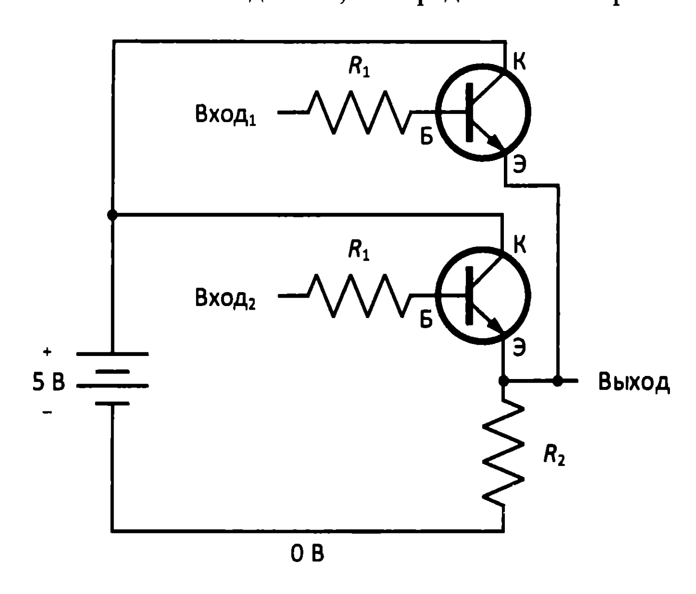


***Рис. П2.** Схема вентиля ИЛИ*

Упражнение 3

Найдите в Интернете бесплатные программные пакеты для разработки на языке VHDL, включающие имитатор. Установите один из этих пакетов, настройте его и попробуйте создать любые простые демонстрационные проекты, включенные в состав пакета, чтобы убедиться, что он работает должным образом.

Ответ. Ниже приведены некоторые бесплатные пакеты для разработки на языке VHDL:

1. Пакет Xilinx Vivado Design доступен по адресу **https://www.xilinx.com/support/download.html**.
2. Пакет Intel® Quartus® Prime Software Lite Edition доступен по адресу **https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html**.
3. GHDL, пакет моделирования для VHDL с открытым исходным кодом доступен по адресу **https://github.com/ghdl/ghdl**.
4. Пакет Mentor ModelSim PE Student Edition доступен по адресу **https://www.mentor.com/company/higher_ed/modelsim-student-edition**.
5. Пакет Electronic Design Automation (EDA) Playground доступен по адресу **https://www. edaplayground.com/**.

В примерах для *главы 2* и следующих глав будет использоваться пакет **Vivado Design Suite**, в том числе для загрузки схемных решений в недорогую плату разра-

ботки ПЛИС. Ниже приведена процедура установки и настройки этого пакета для Windows 10:

1. Посетите веб-сайт **https://www.xilinx.com/support/download.html** и выберите веб-установщик для последней версии Vivado Design Suite for Windows. Необходимо выбрать полный установщик Vivado, а не обновление. При этом вам потребуется создать учетную запись Xilinx, если у вас ее еще нет. Обязательно сохраните имя пользователя и пароль своей учетной записи для использования в дальнейшем.
2. Введите запрошенную информацию, скачайте установщик **Windows Self Extracting Web Installer** и запустите его. Возможно, вам потребуется изменить настройки для установки приложений Windows, чтобы разрешить запуск программы установки.
3. Вам будет предложено войти в систему, используя данные вашей учетной записи Xilinx, и подтвердить согласие с условиями лицензионных соглашений.
4. Выберите пакет инструментов, который вы хотите установить. Для примеров в этой книге используется Vivado. Выберите **Vivado** и щелкните **Next** (Далее).
5. Выберите **Vivado HL WebPack** (это бесплатная версия). Щелкните **Next** (Далее).
6. Подтвердите согласие с предложенной по умолчанию конфигурацией инструментов проектирования, устройств и параметров установки для Vivado HL Webpack. Щелкните **Next** (Далее).
7. Подтвердите согласие с предложенным по умолчанию каталогом установки и другими параметрами. Щелкните **Next** (Далее).
8. На странице **Installation Summary** (Сводка результатов установки), щелкните **Install** (Установить). Загрузка и установка займут некоторое время. Требуемое для этого время зависит от скорости вашего интернет-соединения. Запланируйте ожидание на несколько часов.

После завершения установки выполните следующие действия, чтобы создать пример проекта:

1. На рабочем столе найдите значок с именем, похожим на **Vivado 2021.2**. Дважды щелкните на этом значке (а не на значке с надписью **Vivado HLS**), чтобы запустить данное приложение.
2. В главном окне Vivado нажмите **Open Example Project** (Открыть пример проекта).
3. Перейдите к окну **Select Project Template** (Выбор шаблона проекта) и выберите **CPU (HDL)**.
4. На следующих экранах просмотрите и подтвердите согласие со значениями по умолчанию и щелкните **Finish** (Готово), чтобы создать проект.
5. На странице **Project Manager** (Диспетчер проектов) вы найдете панель **Sources** (Источники). Разверните древовидный список и дважды щелкните на некоторых файлах, чтобы открыть их в редакторе. Большинство файлов в этом примере проекта написаны на языке описания аппаратных средств Verilog.

6. Щелкните **Run Synthesis** (Запустить синтез) на панели **Project Manager** (Диспетчер проектов). По мере выполнения синтеза на панели **Design Runs** (Выполнение проекта) будет обновляться информация о состоянии. Это может занять несколько минут.
7. После завершения синтеза отображается диалоговое окно с предложением запустить реализацию. Щелкните **Cancel** (Отмена).
8. Щелкните **Run Simulation** (Запустить моделирование) в разделе **Project Manager** (Диспетчер проектов) главного окна Vivado, затем выберите **Run behavioral simulation** (Запустить моделирование поведения). Это также может занять несколько минут.
9. После завершения моделирования в окне **Simulation** (Моделирование) появится временна́я диаграмма, отображающая сигналы ЦП, смоделированные с использованием входных данных, предоставленных в исходных файлах моделирования.
10. На этом упражнение завершается. Вы можете закрыть Vivado.

Упражнение 4

Используя набор инструментов VHDL, создайте 4-разрядный сумматор с помощью листингов, представленных в *главе 2*.

Ответ. Для создания 4-разрядного сумматора выполните следующие действия:

1. Дважды щелкните на значке **Vivado 2021.2** (или подобном), чтобы запустить Vivado.
2. Щелкните **Create Project** (Создать проект) в главном окне Vivado.
3. С помощью мыши просмотрите и примите предложенные по умолчанию название и местоположение проекта.
4. Выберите **RTL Project** — тип проекта по умолчанию.
5. На странице **Default Part** (Часть по умолчанию) выберите **Boards** (Платы). Введите `Arty` в поле поиска, выберите **Arty A7-35**, затем щелкните **Next** (Далее). Если в результате поиска **Arty** не отображается, щелкните **Update Board Repositories** (Обновить репозитории плат) и выполните поиск снова.
6. Нажмите **Finish** (Готово) для создания проекта.
7. На панели **Project Manager** (Диспетчер проектов) щелкните **Add Sources** (Добавить источники), выберите **Add or create design sources** (Добавить или создать источники проекта), добавьте файлы `Ex__4_adder4.vhdl` и `Ex__4_fulladder.vhdl`, затем щелкните **Finish** (Готово).
8. В окне **Project Manager** (Диспетчер проектов) разверните дерево окна **Design Sources** (Источники проекта) и найдите два добавленных вами файла. Дважды щелкните на каждом из них и разверните окно исходного кода, чтобы просмотреть код.

9. Щелкните **Run Synthesis** (Запустить синтез) на панели **Project Manager** (Диспетчер проектов). Оставьте для параметров в окне **Launch Runs** (Выполнение запуска) значения по умолчанию и нажмите **OK**. По мере выполнения синтеза на панели **Design Runs** (Выполнение проекта) будет обновляться информация о состоянии.
10. Дождитесь завершения синтеза, затем в окне **Synthesis Completed** (Синтез завершен) выберите **View Reports** (Просмотр отчетов). Дважды щелкните на некоторых отчетах, созданных в процессе синтеза. В наличии имеются только те отчеты, значок которых помечен зеленой точкой.
11. На этом упражнение завершается. Вы можете закрыть Vivado.

Упражнение 5

Добавьте в свой 4-разрядный сумматор код тестового драйвера (примеры можно найти в Интернете по фразе *"VHDL testbench"*), затем запустите его с ограниченным набором входных данных и проверьте правильность выходных данных.

Ответ. Выполните следующие действия для тестирования проекта 4-разрядного сумматора, созданного в *упражнении 4*:

1. Дважды щелкните на значке **Vivado 2021.2** (или подобном), чтобы запустить Vivado.
2. Щелкните **Open Project** (Открыть проект) в главном окне Vivado и откройте проект, который вы создали в *упражнении 4*. Необходимо выбрать файл проекта с расширением `.xpr`.
3. На панели **Project Manager** (Диспетчер проектов) щелкните **Add Sources** (Добавить источники), выберите **Add or create simulation sources** (Добавление или создание источников моделирования), добавьте файл `Ex__5_adder4_testbench.vhdl`, затем щелкните **Finish** (Готово).
4. В окне **Project Manager** (Диспетчер проектов) разверните дерево окна **Design Sources** (Источники проекта) и найдите добавленный вами файл. Дважды щелкните на этом файле и разверните окно исходного кода, чтобы просмотреть код. Обратите внимание на шесть тестовых сценариев, присутствующих в коде.
5. Щелкните **Run Simulation** (Запустить моделирование) в разделе **Project Manager** (Диспетчер проектов) главного окна Vivado, затем выберите **Run behavioral simulation** (Запустить моделирование поведения).
6. Дождитесь завершения моделирования, затем разверните окна с временно́й диаграммой (возможно, обозначенной **Untitled 1**).
7. Используйте значок увеличительного стекла и горизонтальную полосу прокрутки окна, чтобы просмотреть шесть тестовых сценариев за первые 60 **наносекунд (нс)** выполнения. Определите, правильно ли указаны сумма и признак переноса для каждой операции сложения. Для того чтобы обновить информацию в столбце **Value** (Значение), можно перетащить желтый маркер.

8. На этом упражнение завершается. Вы можете закрыть Vivado.

Код набора тестов содержится в VHDL-файле `Ex__5_adder4_testbench.vhdl`:

```
library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;

entity ADDER4_TESTBENCH is
end entity ADDER4_TESTBENCH;

architecture BEHAVIORAL of ADDER4_TESTBENCH is

  component ADDER4 is
    port (
      A4       : in    std_logic_vector(3 downto 0);
      B4       : in    std_logic_vector(3 downto 0);
      SUM4     : out   std_logic_vector(3 downto 0);
      C_OUT4   : out   std_logic
    );
  end component;

  signal a     : std_logic_vector(3 downto 0);
  signal b     : std_logic_vector(3 downto 0);
  signal s     : std_logic_vector(3 downto 0);
  signal c_out : std_logic;

begin

  TESTED_DEVICE : ADDER4
    port map (
      A4     => a,
      B4     => b,
      SUM4   => s,
      C_OUT4 => c_out
    );

  TEST : process
  begin
    a <= "0000";
    b <= "0000";
```

```
    wait for 10 ns;
    a <= "0110";
    b <= "1100";

    wait for 10 ns;
    a <= "1111";
    b <= "1100";

    wait for 10 ns;
    a <= "0110";
    b <= "0111";

    wait for 10 ns;
    a <= "0110";
    b <= "1110";

    wait for 10 ns;
    a <= "1111";
    b <= "1111";

    wait;

  end process TEST;

end architecture BEHAVIORAL;
```

Упражнение 6

Раскройте код тестового драйвера и убедитесь, что 4-разрядный сумматор выдает правильные результаты для всех возможных комбинаций входных данных.

Ответ. Выполните следующие действия для тестирования проекта 4-разрядного сумматора, созданного в *упражнении 4*:

1. Дважды щелкните на значке **Vivado 2021.2** (или подобном), чтобы запустить Vivado.
2. Щелкните **Open Project** (Открыть проект) в главном окне Vivado и откройте проект, который вы создали в *упражнении 4* и изменили в *упражнении 5*. Необходимо выбрать файл проекта с расширением `.xpr`.
3. Мы собираемся заменить код тестового драйвера в *упражнении 5* другим кодом тестового драйвера. В окне **Project Manager** (Диспетчер проектов) разверните дерево окна **Simulation Sources** (Источники моделирования) и найдите модуль,

который был добавлен в *упражнении 5* (ADDER4_TESTBENCH). Щелкните правой кнопкой мыши на имени модуля, выберите **Remove File from Project** (Удалить файл из проекта), затем щелкните **OK**, чтобы подтвердить удаление.

4. На панели **Project Manager** (Диспетчер проектов) щелкните **Add Sources** (Добавить источники), выберите **Add or create simulation sources** (Добавление или создание источников моделирования), добавьте файл `Ex__6_adder4_fulltestbench.vhdl`, затем щелкните **Finish** (Готово).
5. В окне **Project Manager** (Диспетчер проектов) разверните дерево окна **Design Sources** (Источники проекта) и найдите добавленный вами файл. Дважды щелкните на этом файле и разверните окно исходного кода, чтобы просмотреть код. Обратите внимание на размещенный в коде цикл с 256 тестовыми сценариями.
6. Щелкните **Run Simulation** (Запустить моделирование) в разделе **Project Manager** (Диспетчер проектов) главного окна Vivado, затем выберите **Run behavioral simulation** (Запустить моделирование поведения).
7. Дождитесь завершения моделирования, затем разверните окна с временнóй диаграммой (возможно, обозначенной **Untitled 1**).
8. Для просмотра тестовых сценариев используйте значок увеличительного стекла и горизонтальную полосу прокрутки окна. *Ой!* Выполнение останавливается через 1000 нс — этого недостаточно для выполнения всех тестов.
9. Щелкните правой кнопкой мыши на **Simulation** (Моделирование) в панели **Project Manager** (Диспетчер проектов), затем выберите **Simulation Settings...** (Настройки моделирования...).
10. Выберите вкладку **Simulation** (Моделирование) и измените значение параметра `xsim.simulate.runtime` на `3000ns`. Щелкните **OK**.
11. Щелкните × в окне **Simulation** (Моделирование), чтобы закрыть его.
12. Снова запустите процесс моделирования.
13. После раскрытия и масштабирования временнóй диаграммы вы сможете увидеть все 256 тестовых сценариев. Проверьте, есть ли на графике места, где сигнал ошибки имеет значение 1. Это указывало бы на то, что выходные данные сумматора не соответствуют ожидаемым выходным данным.
14. На этом упражнение завершается. Вы можете закрыть Vivado.

Код набора тестов содержится в VHDL-файле `Ex__6_adder4_fulltestbench.vhdl`:

```
library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
  use IEEE.NUMERIC_STD.ALL;

entity ADDER4_TESTBENCH is
end entity ADDER4_TESTBENCH;
```

```
architecture BEHAVIORAL of ADDER4_TESTBENCH is

  component ADDER4 is
    port (
      A4        : in    std_logic_vector(3 downto 0);
      B4        : in    std_logic_vector(3 downto 0);
      SUM4      : out   std_logic_vector(3 downto 0);
      C_OUT4    : out   std_logic
    );
  end component;

  signal a              : std_logic_vector(3 downto 0);
  signal b              : std_logic_vector(3 downto 0);
  signal s              : std_logic_vector(3 downto 0);
  signal c_out          : std_logic;

  signal expected_sum5 : unsigned(4 downto 0);
  signal expected_sum4 : unsigned(3 downto 0);
  signal expected_c    : std_logic;
  signal error         : std_logic;

begin
  TESTED_DEVICE : ADDER4
    port map (
      A4     => a,
      B4     => b,
      SUM4   => s,
      C_OUT4 => c_out
    );

  TEST : process
  begin

    -- Протестируем все комбинации двух 4-разрядных слагаемых (всего 256 тестов)
    for a_val in 0 to 15 loop
      for b_val in 0 to 15 loop
        -- Установим входные данные для компонента ADDER4
        a <= std_logic_vector(to_unsigned(a_val, a'length));
        b <= std_logic_vector(to_unsigned(b_val, b'length));
        wait for 1 ns;
```

```
      -- Вычислим 5-битную сумму двух 4-битных значений
      expected_sum5 <= unsigned('0' & a) + unsigned('0' & b);
      wait for 1 ns;

      -- Разобьем сумму на 4-разрядное выходное значение и бит переноса
      expected_sum4 <= expected_sum5(3 downto 0);
      expected_c    <= expected_sum5(4);
      wait for 1 ns;

      -- Сигнал 'ошибка' принимает значение 1 только при обнаружении ошибки
      if ((unsigned(s) = unsigned(expected_sum4)) and
          (c_out = expected_c)) then
        error <= '0';
      else
        error <= '1';
      end if;

      -- Каждый проход через внутренний цикл занимает 10 нс
      wait for 7 ns;

    end loop;
  end loop;

  wait;

end process TEST;

end architecture BEHAVIORAL;
```

Глава 3. Элементы процессора

Упражнение 1

Рассмотрим сложение двух 8-битных чисел со знаком (т. е. чисел в диапазоне от –128 до +127), где один операнд положительный, а другой — отрицательный. Существует ли какая-либо пара 8-битных чисел с разными знаками, сумма которых выходит за пределы диапазона от –128 до +127? Такая ситуация будет представлять собой знаковое переполнение. Примечание: мы рассматриваем здесь только сложение, потому что, как мы видели, вычитание в архитектуре процессора 6502 — это то же самое, что и сложение, но с инверсией битов правого операнда.

Ответ. Диапазон положительных (или неотрицательных) чисел — от 0 до 127. Диапазон отрицательных чисел — от –128 до –1. Для того чтобы охватить все возможности, необходимо рассмотреть только крайние значения каждого из этих диапазонов:

Сумма	Результат
0 + –128	–128
127 + –128	–1
0 + –1	–1
127 + –1	126

Из предыдущей таблицы видно, что не существует пары 8-битных чисел с разными знаками, сумма которых выходит за пределы диапазона от –128 до +127.

Упражнение 2

Если ответ на *упражнение 1* — *"нет"*, это означает, что единственный способ создать знаковое переполнение — сложить два числа с одинаковыми знаками. Если происходит переполнение, что вы можете сказать о результате выполнения операции "исключающее ИЛИ" между старшим битом каждого операнда и старшим битом результата? Другими словами, каков будет результат выражений `left(7) XOR result(7)` и `right(7) XOR result(7)`? В этих выражениях `(7)` указывает на бит 7 — старший бит.

Ответ. Бит 7 — это знаковый бит. Поскольку переполнение может произойти только тогда, когда оба операнда имеют одинаковый знак, то когда происходит переполнение, `left(7)` должно быть равно `right(7)`.

Когда происходит переполнение, знак результата отличается от знака двух операндов. Это означает, что `result(7)` отличается от бита 7 обоих операндов.

Поэтому всякий раз, когда происходит переполнение, `left(7) XOR result(7) = 1` и `right(7) XOR result(7) = 1`.

Упражнение 3

Посмотрите на листинг VHDL в *разд. "Арифметико-логическое устройство" главы 3* и определите, является ли логика установки и обнуления флага V корректной для операций сложения и вычитания. Проверьте результаты сложения 126 + 1, 127 + 1, –127 + (–1) и –128 + (–1).

Ответ. Приведенный в *главе 3* листинг реализации части **арифметико-логического устройства** (АЛУ) на языке VHDL, подобного тому, что применяется в процессоре 6502, выполняет вычисление флага переполнения с помощью следующего кода:

```
if (((LEFT(7) XOR result8(7)) = '1') AND
    ((right_op(7) XOR result8(7)) = '1')) then -- V flag
  V_OUT <= '1';
else
  V_OUT <= '0';
end if;
```

В следующей таблице приведены результаты выполнения этого кода для четырех тестовых сценариев, указанных в вопросе:

left	right	left(7)	right(7)	result8(7)	V_OUT	Правильно?
126	1	0	0	0	0	Да
127	1	0	0	1	1	Да
–127	–1	1	1	1	0	Да
–128	–1	1	1	0	1	Да

Логика установки и обнуления флага `V` для этих тестовых сценариев верна.

Упражнение 4

При пересылке данных через среду передачи, подверженную ошибкам, чтобы определить, были ли какие-либо биты данных потеряны или искажены во время передачи, обычно используется **контрольная сумма**. Она, как правило, добавляется к переданной записи данных. В одном из алгоритмов расчета контрольной суммы используются следующие шаги:

1. Сложение всех байтов в записи данных с сохранением только младших 8 бит суммы.
2. Определение контрольной суммы, которая представляет собой дополнение до двух этой 8-битной суммы.
3. Добавление байта контрольной суммы к записи данных.

После получения блока данных с добавленной контрольной суммой процессор может определить правильность контрольной суммы простым сложением всех байтов в записи, включая контрольную сумму. Контрольная сумма является правильной, если младшие 8 бит суммы равны нулю. Реализуйте этот алгоритм контрольной суммы, используя язык ассемблера 6502. Байты данных начинаются с ячейки памяти по адресам `$10`–`$11`, а количество байтов (включая байт контрольной суммы) предоставляется в качестве входных данных в регистре X. Установите для регистра A значение 1, если контрольная сумма является правильной, и значение 0 в ином случае.

Ответ. Файл `Ex__4_checksum_alg.asm` содержит следующий код вычисления контрольной суммы:

```
; Ex__4_checksum_alg.asm
; Попробуйте запустить этот код по адресу https://skilldrick.github.io/easy6502/

; Зададим массив байтов для вычисления контрольной суммы
LDA #$01
STA $00

LDA #$72
STA $01

LDA #$93
STA $02

LDA #$F4
STA $03

LDA #$06 ; Это байт контрольной суммы
STA $04

; Сохраним адрес массива данных в $10-$11
LDA #$00
STA $10
STA $11

; Сохраним количество байтов в X
LDX #5

; Зададим алгоритм вычисления контрольной суммы.
; Переместим содержимое X в Y
TXA
TAY

; Вычислим контрольную сумму
LDA #$00
DEY

LOOP:
CLC
```

```
ADC ($10), Y
DEY
BPL LOOP

CMP #$00
BNE ERROR

; Сумма равна нулю: контрольная сумма верна
LDA #1
JMP DONE

; Сумма не равна нулю: контрольная сумма неверна
ERROR:
LDA #0

; A содержит 1, если контрольная сумма верна, и 0, если она неверна
DONE:
```

Упражнение 5

Поместите код проверки контрольной суммы из *упражнения 4* в помеченную подпрограмму, которую можно вызвать с помощью инструкции JSR и которая завершается инструкцией RTS.

Ответ. Файл Ex__5_checksum_subroutine.asm реализует алгоритм вычисления контрольной суммы в виде подпрограммы:

```
; Ex__5_checksum_subroutine.asm
; Попробуйте запустить этот код по адресу https://skilldrick.github.io/easy6502/

; Зададим массив байтов для вычисления контрольной суммы
LDA #$01
STA $00
LDA #$72
STA $01
LDA #$93
STA $02
LDA #$F4
STA $03
LDA #$06 ; Это байт контрольной суммы
STA $04
```

```
; Сохраним адрес массива данных в $10-$11
LDA #$00
STA $10
STA $11

; Сохраним количество байтов в X
LDX #5

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

; Остановим выполнение
BRK

; ============================================
; Вычислим контрольную сумму
CALC_CKSUM:
; Переместим содержимое X в Y
TXA
TAY

LDA #$00
DEY

LOOP:
CLC
ADC ($10), Y
DEY
BPL LOOP

CMP #$00
BNE CKSUM_ERROR

; Сумма равна нулю: контрольная сумма верна
LDA #1
JMP DONE

; Сумма не равна нулю: контрольная сумма неверна
CKSUM_ERROR:
LDA #0
```

```
; A содержит 1, если контрольная сумма верна, и 0, если она неверна

DONE:
RTS
```

Упражнение 6

Напишите и выполните набор тестов для проверки правильности работы подпрограммы проверки контрольной суммы, которую вы создали в *упражнениях 4–5*. Какова длина самого короткого блока данных, для которого ваш код может выполнить проверку контрольной суммы? Какова длина самого длинного блока данных?

Ответ. Файл `Ex__6_checksum_tests.asm` содержит следующий код проверки контрольной суммы:

```
; Ex__6_checksum_tests.asm
; Попробуйте запустить этот код по адресу https://skilldrick.github.io/easy6502/

; После завершения тестов: A=$AA, если проверка успешна;
; A=$EE, если обнаружена ошибка

; Сохраним адрес массива данных в $10-$11
LDA #$00
STA $10
STA $11

TEST1:

; ==============================================
; Тест 1: 1 байт; контрольная сумма: 00. Тест должен быть пройден? Да
LDA #$00
STA $00

; Сохраним количество байтов в X
LDX #1

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$01
BEQ TEST2
JMP ERROR
```

```
TEST2:

; ==============================================
; Тест 2: 1 байт; контрольная сумма: 01. Тест должен быть пройден? Нет
LDA #$01
STA $00

; Сохраним количество байтов в X
LDX #1

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$00
BEQ TEST3
JMP ERROR

TEST3:
; ==============================================
; Тест 3: 2 байта: 00; контрольная сумма: 00. Тест должен быть пройден? Да
LDA #$00
STA $00
STA $01

; Сохраним количество байтов в X
LDX #2

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$01
BEQ TEST4
JMP ERROR

TEST4:
; ==============================================
; Тест 4: 2 байта: 00; контрольная сумма: 01. Тест должен быть пройден? Нет
LDA #$00
STA $00
```

```
LDA #$01
STA $01

; Сохраним количество байтов в X
LDX #2

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$00
BEQ TEST5
JMP ERROR

TEST5:
; ================================================
; Тест 5: 2 байта: 01; контрольная сумма: 00. Тест должен быть пройден? Нет
LDA #$01
STA $00
LDA #$00
STA $01

; Сохраним количество байтов в X
LDX #1

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$00
BEQ TEST6
JMP ERROR

TEST6:
; ================================================
; Тест 6: 3 байта: 00 00; контрольная сумма: 00. Тест должен быть пройден? Да
LDA #$00
STA $00
STA $01

STA $02
```

```
; Сохраним количество байтов в X
LDX #3

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$01
BEQ TEST7
JMP ERROR

TEST7:
; ============================================
; Тест 7: 3 байта: 00 00; контрольная сумма: 00. Тест должен быть пройден? Да
LDA #$00
STA $00
STA $01
STA $02

; Сохраним количество байтов в X
LDX #3

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$01
BEQ TEST8
JMP ERROR

TEST8:
; ============================================
; Тест 8: 3 байта: 00 00; контрольная сумма: 01. Тест должен быть пройден? Нет
LDA #$00
STA $00
LDA #$00
STA $01

LDA #$01
STA $02

; Сохраним количество байтов в X
LDX #3
```

```
; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$00
BEQ TEST9
JMP ERROR

TEST9:
; ==============================================
; Тест 9: 3 байта: 00 01; контрольная сумма: FF. Тест должен быть пройден? Да
LDA #$00
STA $00
LDA #$01
STA $01
LDA #$FF
STA $02

; Сохраним количество байтов в X
LDX #3

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$01
BEQ TEST10
JMP ERROR

TEST10:
; ==============================================
; Тест 10: 5 байтов: 01 72 93 F4; контрольная сумма: 06
; Тест должен быть пройден? Да
LDA #$01

STA $00
LDA #$72
STA $01
LDA #$93
STA $02
LDA #$F4
STA $03
```

```
LDA #$06 ; Это байт контрольной суммы
STA $04

; Сохраним количество байтов в X
LDX #5

; Вызовем подпрограмму вычисления контрольной суммы
JSR CALC_CKSUM

CMP #$01
BEQ PASSED

ERROR:
; ==============================================
; Произошла ошибка; останов выполнения со значением $EE в A
LDA #$EE
BRK

PASSED:
; ==============================================
; Все тесты пройдены; останов выполнения со значением $AA в A
LDA #$AA
BRK

; ==============================================
; Вычислим контрольную сумму
CALC_CKSUM:
; Переместим содержимое X в Y
TXA
TAY

LDA #$00
DEY

LOOP:
CLC
ADC ($10), Y
DEY
BPL LOOP

CMP #$00
BNE CKSUM_ERROR
```

```
; Сумма равна нулю: контрольная сумма верна
LDA #1
JMP DONE

; Сумма не равна нулю: контрольная сумма неверна
CKSUM_ERROR:
LDA #0

; A содержит 1, если контрольная сумма верна, и 0, если она неверна
DONE:
RTS
```

Данная процедура вычисления контрольной суммы работает для последовательностей длиной от 1 до 255 байт.

Глава 4. Компоненты компьютерной системы

Упражнение 1

Создайте схему реализации логического элемента И-НЕ, используя две пары МОП-транзисторов, объединенных в КМОП-структуры. В отличие от схем вентилей на основе *n-p-n*-транзисторов, для этой схемы не требуются резисторы.

Ответ. Эта схема выглядит так, как представлено на рис. П3.

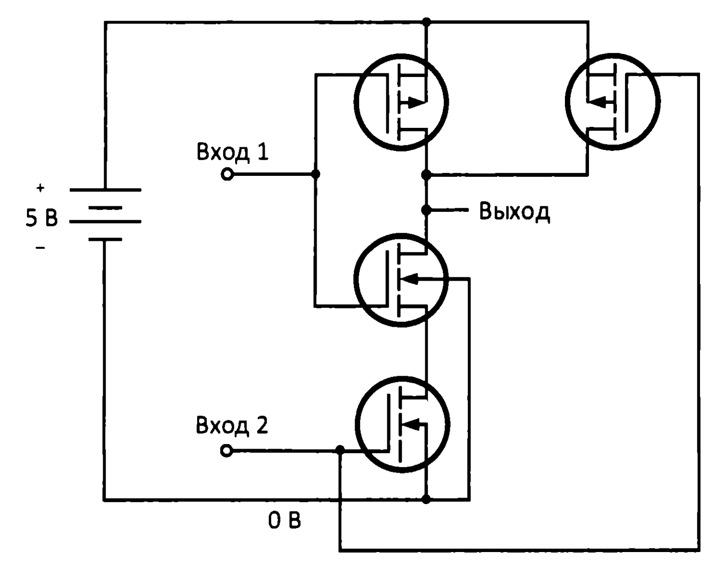


Рис. П3. *Схема вентиля И-НЕ*

Упражнение 2

16-гигабитная интегральная схема памяти DRAM имеет два входа выбора группы банков, два входа выбора банка и 17 входов выбора адресов строк. Сколько битов в каждой строке банка в этом устройстве?

Ответ. Эта схема памяти DRAM содержит 16 гигабит = 16×2^{30} бит.

Количество битов адреса = 2 бита группы банков + 2 бита банка + 17 битов адреса строки = 21 бит. Следовательно, размер строки каждого банка равен $(16 \times 2^{30}) : 2^{21}$ = **8192 бита**.

Глава 5. Аппаратно-программный интерфейс

Упражнение 1

Перезагрузите компьютер и войдите в раздел настроек BIOS или UEFI. Изучите каждое из доступных в этом разделе меню. Какую систему использует ваш компьютер, BIOS или UEFI? Поддерживает ли ваша материнская плата разгон процессора? Когда закончите, обязательно выберите вариант выхода без сохранения изменений, если только вы не обладаете полной уверенностью в том, что хотите внести изменения.

Ответ. В Windows вы можете войти в раздел настроек BIOS/UEFI путем изменения параметров запуска во время работы Windows. Для того чтобы получить доступ к этим настройкам, выполните следующие действия:

1. В окне поиска Windows введите `startup` и выберите **Change advanced startup options** (Изменение расширенных параметров запуска).
2. Щелкните на кнопке **Restart now** (Перезагрузить сейчас) в разделе **Advanced startup** (Особые варианты загрузки).
3. При появлении запроса **Choose an option** (Выберите вариант) выберите **Troubleshoot** (Устранение неполадок).
4. На экране **Troubleshoot** (Устранение неполадок) выберите **Advanced options** (Дополнительные параметры).
5. На экране **Advanced options** (Дополнительные параметры) выберите **UEFI Firmware Settings** (Параметры встроенного ПО UEFI).
6. На экране **UEFI Firmware Settings** (Параметры встроенного ПО UEFI) щелкните на кнопке **Restart** (Перезапустить).
7. Система перезагружается и отображает главный экран настройки UEFI. Используйте клавиши со стрелками влево и вправо на клавиатуре для перемещения между экранами.

Следующие пункты содержат ответы на вопросы этого упражнения для конкретной компьютерной системы (в данном случае это ноутбук Asus ZenBook UX303LA):

1. В сообщениях, отображаемых в меню, часто используется термин "BIOS", однако упоминания о "приложениях EFI" и датах их выпуска указывают на то, что на самом деле это **UEFI**.
2. Никакие варианты разгона **не предусмотрены**.

После завершения изучения информации UEFI выполните выход без сохранения каких-либо изменений с помощью следующих действий:

1. Перейдите на страницу **Save & Exit** (Сохранить и закрыть).
2. Используйте клавиши со стрелками вверх и вниз, чтобы выбрать вариант **Discard Changes and Exit** (Отменить изменения и выйти).
3. Нажмите клавишу <Enter>.
4. Выберите **Yes** (Да) и нажмите клавишу <Enter> при отображении окна **Exit Without Saving** (Выход без сохранения).
5. Система перезагружается.

Упражнение 2

Выполните соответствующую команду на компьютере, чтобы отобразить информацию о текущих запущенных процессах. Какой идентификатор **Process ID (PID)** имеет процесс, который вы используете для выполнения этой команды?

Ответ. В Windows откройте окно командной строки (чтобы найти это приложение, введите `command` в строке поиска Windows) и введите команду `tasklist`:

```
C:\>tasklist

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0          8 K
System                           4 Services                   0      9,840 K
Registry                       120 Services                   0     85,324 K
smss.exe                       544 Services                   0        640 K
csrss.exe                      768 Services                   0      4,348 K
wininit.exe                    852 Services                   0      4,912 K
services.exe                   932 Services                   0      8,768 K
lsass.exe                      324 Services                   0     18,160 K
svchost.exe                   1044 Services                   0      2,308 K
svchost.exe                   1068 Services                   0     27,364 K
```

```
.
.
.
svchost.exe                   12184 Services                   0      8,544 K
cmd.exe                       16008 Console                    3      3,996 K
conhost.exe                   21712 Console                    3     18,448 K
tasklist.exe                  15488 Console                    3     10,096 K
```

Текущий процесс — это тот, который запускает приложение `tasklist.exe`. Идентификатор (PID) этого процесса — `15488`.

Глава 6. Специализированные вычисления

Упражнение 1

Частотно-монотонное планирование (RMS) — это алгоритм назначения приоритетов потокам в приложениях жесткого реального времени с вытеснением, в которых потоки выполняются периодически. RMS назначает наивысший приоритет потоку с самым коротким периодом выполнения, следующий по значимости приоритет — потоку со следующим наиболее коротким периодом выполнения и т. д. Система RMS является диспетчеризуемой, т. е. все ее задачи гарантированно укладываются в установленные сроки (при условии, что межпоточные взаимодействия или другие действия, такие как прерывания, не вызывают задержек обработки), если выполняется следующее условие:

$$\sum_{i=1}^{n}\frac{C_i}{T_i} \le n\left(2^{1/n}-1\right).$$

Эта формула выражает максимальную долю доступного времени обработки, которое может быть потрачено n потоками. В этой формуле C_i — это максимальное время выполнения, необходимое для потока i, а T_i — период выполнения потока i.

Является ли следующая система, состоящая из трех потоков, диспетчеризуемой?

Поток	Время выполнения C_i, мс	Период выполнения T_i, мс
Поток 1	50	100
Поток 2	100	500
Поток 3	120	1000

Ответ. Прежде всего оценим левую часть формулы RMS, используя данные из таблицы:

$$\frac{50}{100}+\frac{100}{500}+\frac{120}{1000}=0{,}82.$$

Затем оценим правую часть формулы RMS:

$$3\cdot\left(2^{1/3}-1\right)=0{,}7798.$$

Так как 0,82 не меньше и не равно 0,7798, этот набор задач не является диспетчеризуемым в системе RMS.

Упражнение 2

Широко используемая форма одномерного **дискретного косинусного преобразования (ДКП)** выражается следующей формулой:

$$X_k=\sum_{n=0}^{N-1}x_n\cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right].$$

В этой формуле k, индекс коэффициента ДКП, меняется от 0 до $N-1$. Напишите программу для вычисления ДКП такой последовательности:

x = {0.5, 0.2, 0.7, -0.6, 0.4, -0.2, 1.0, -0.3}.

Косинусные члены в формуле зависят только от индексов n и k и не зависят от последовательности входных данных x. Это означает, что косинусные члены можно вычислить один раз и сохранить в виде констант для последующего использования. Если сделать это в качестве подготовительного этапа, то вычисление каждого коэффициента ДКП сводится к последовательности операций MAC.

Эта формула представляет собой неоптимизированную форму вычисления ДКП, требующую N^2 итераций операции MAC для вычисления всех коэффициентов ДКП в количестве N.

Ответ. Файл Python `Ex__2_dct_formula.py` содержит код вычисления ДКП:

```
#!/usr/bin/env python

"""Ex__2_dct_formula.py: ответ на упражнение 2 главы 6."""

# Выходные данные, формируемые этой программой:
# Индекс      0       1       2       3       4       5       6       7
# x      0.5000  0.2000  0.7000 -0.6000  0.4000 -0.2000  1.0000 -0.3000
# DCT(x) 1.7000  0.4244  0.6374  0.4941 -1.2021  0.5732 -0.4936  2.3296

import math
```

```
# Входной вектор
x = [0.5, 0.2, 0.7, -0.6, 0.4, -0.2, 1.0, -0.3]

# Вычислим коэффициенты ДКП
dct_coef = [[i for i in range(len(x))] for j in range(len(x))]
for n in range(len(x)):
    for k in range(len(x)):
        dct_coef[n][k] = math.cos((math.pi/len(x))*(n + 1/2)*k)

# Вычислим ДКП
x_dct = [i for i in range(len(x))]
for k in range(len(x)):
    x_dct[k] = 0;
    for n in range(len(x)):
        x_dct[k] += x[n]*dct_coef[n][k]

# Выведем результаты
print('Индекс', end='')
for i in range(len(x)):
    print("%8d" % i, end='')

print('\nx      ', end='')
for i in range(len(x)):
    print("%8.4f" % x[i], end='')

print('\nDCT(x) ', end='')
for i in range(len(x)):
    print("%8.4f" % x_dct[i], end='')
```

Для того чтобы запустить код при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__2_dct_formula.py
```

Ниже показаны выходные данные, формируемые программой:

```
C:\>Ex__2_dct_formula.py
Индекс       0       1       2       3       4       5       6       7
x       0.5000  0.2000  0.7000 -0.6000  0.4000 -0.2000  1.0000 -0.3000
DCT(x)  1.7000  0.4244  0.6374  0.4941 -1.2021  0.5732 -0.4936  2.3296
```

Упражнение 3

В качестве функции активации в **искусственных нейронных сетях** (**artificial neural network, ANN**) часто используется гиперболический тангенс. Его функция определяется следующим образом:

$$\mathrm{th}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Дан нейрон с входами от трех предшествующих нейронов, показанный на рис. 6.4. Вычислите выходной сигнал нейрона с гиперболическим тангенсом в качестве функции активации $F(x)$, используя следующие выходные сигналы нейронов и вeсá соответствующих путей:

Нейрон	Выходной сигнал нейрона	Вес
N_1	0,6	0,4
N_2	–0,3	0,8
N_3	0,5	–0,2

Ответ. Файл Python `Ex__3_activation_func.py` содержит следующий код:

```
#!/usr/bin/env python

"""Ex__3_activation_func.py: ответ на упражнение 3 главы 6."""

# Выходные данные, формируемые этой программой:
# Выходное значение нейрона = -0.099668

import math

# Векторы сигнала и веса нейрона
neuron = [0.6, -0.3,  0.5]
weight = [0.4,  0.8, -0.2]

sum = 0
for i in range(len(neuron)):
    sum = sum + neuron[i] * weight[i]

output = math.tanh(sum)

# Вывод результатов
print('Выходной сигнал нейрона = %8.6f' % output)
```

Для того чтобы запустить код при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__3_activation_func.py
```

Ниже показаны выходные данные, формируемые программой:

```
C:\>Ex__3_activation_func.py
Выходной сигнал нейрона = -0.099668
```

Глава 7. Архитектура процессоров и памяти

Упражнение 1

16-разрядный встраиваемый процессор имеет отдельные области памяти для кода и данных. Код хранится во флеш-памяти, а изменяемые данные — в оперативной памяти. Некоторые значения данных, такие как константы и начальные значения для элементов данных оперативной памяти, хранятся в той же области флеш-памяти, что и инструкции программы. ОЗУ и ПЗУ находятся в одном адресном пространстве. Какая из архитектур процессоров, рассмотренных в *главе 7*, лучше всего описывает этот процессор?

Ответ. Поскольку код и данные расположены в одном и том же адресном пространстве, это **архитектура фон Неймана**.

Тот факт, что код и некоторые элементы данных хранятся в ПЗУ, а другие элементы данных находятся в ОЗУ, не имеет отношения к определению категории архитектуры.

Упражнение 2

Процессор, описанный в *упражнении 1*, имеет функции защиты памяти, которые не позволяют исполняемому коду изменять память инструкций программы. Для доступа к инструкциям и данным этот процессор использует физические адреса. Содержит ли он блок управления памятью (MMU)?

Ответ. Защита областей памяти является функцией MMU, однако наличие защиты памяти само по себе не означает, что используется MMU. *Этот процессор не содержит MMU.*

MMU обычно выполняют трансляцию виртуальных адресов в физические, что в описанном здесь процессоре не предусмотрено.

Упражнение 3

Порядок доступа к последовательным элементам в большой структуре данных может оказать ощутимое влияние на скорость обработки из-за таких факторов, как повторное использование записей буфера TLB. Последовательный доступ к отдаленным элементам массива (это элементы, находящиеся за пределами страничного кадра, где размещены ранее запрошенные элементы) требует частых мягких отказов страниц по мере загрузки новых записей TLB и удаления старых.

Напишите программу, которая создает большой двумерный массив чисел, например на 10 000 строк и 10 000 столбцов. Выполните итеративный обход массива в порядке возрастания столбцов, подписывая каждый элемент суммой индексов строки и столбца. Доступ по столбцам означает, что индекс столбца увеличивается быстрее всего. Другими словами, индекс столбца увеличивается во внутреннем цикле. Точно измерьте время, которое занимает эта процедура. Обратите внимание, что вам может потребоваться принять меры для того, чтобы ваш язык программирования не исключил при оптимизации весь этот расчет из-за того, что результаты работы с массивом не используются в дальнейшем. Может быть достаточно вывести одно из значений массива после завершения отсчета времени или же может потребоваться сделать что-то вроде суммирования всех элементов массива и вывода этого результата.

Повторите процесс, включая измерение времени, точно так же, как было описано ранее, только измените внутренний цикл на итерацию по индексу строки (первый индекс), а внешний цикл — на итерацию по индексу столбца, сделав последовательность доступа построчной.

Во время выполнения вашего кода компьютеры общего назначения выполняют множество других задач, поэтому, чтобы получить статистически достоверный результат, вам может понадобиться выполнить обе процедуры несколько раз. Для начала можно провести эксперимент 10 раз и усреднить время доступа к массиву по столбцам и по строкам.

Можете ли вы определить метод доступа к массиву, постоянно дающий лучший результат? Какой порядок является самым быстрым в вашей системе при использовании выбранного вами языка? Следует учитывать, что разница между методами с порядком доступа по столбцам и по строкам может быть не слишком значительной — она может составлять всего несколько процентов.

Ответ. Файл `Ex__3_row_column_major_order.py` содержит следующую реализацию решения этого упражнения на языке Python:

```
#!/usr/bin/env python

"""Ex__3_row_column_major_order.py: ответ на упражнение 3 главы 7."""

# Типичный результат выполнения этого сценария:
# среднее время доступа по строкам:  16,68 с
```

```
# среднее время доступа по столбцам: 15,94 с
# средняя разница по времени:         0,74 с
# победитель - индексация по столбцам; она быстрее на 4,42%

import time

dim = 10000
matrix = [[0] * dim] * dim

num_passes = 10
row_major_time = 0
col_major_time = 0

for k in range(num_passes):
    print(' Проход %d из %d:' % (k+1, num_passes))

    t0 = time.time()
    for i in range(dim):
        for j in range(dim):

            matrix[i][j] = i + j

    t1 = time.time()

    total_time = t1 - t0
    col_major_time = col_major_time + total_time
    print(' Время заполнения массива по столбцам: %.2f c' %
          total_time)

    t0 = time.time()
    for i in range(dim):
        for j in range(dim):
            matrix[j][i] = i + j

    t1 = time.time()

    total_time = t1 - t0
    row_major_time = row_major_time + total_time
```

```
        print(' Время заполнения массива по строкам: %.2f с' %
              total_time)
        print('')

    row_major_average = row_major_time / num_passes
    col_major_average = col_major_time / num_passes

    if (row_major_average < col_major_average):
        winner = 'строкам'
        pct_better = 100 * (col_major_average -
            row_major_average) / col_major_average
    else:
        winner = 'столбцам'
        pct_better = 100 * (row_major_average -
            col_major_average) / row_major_average

    print('Среднее время доступа по строкам:  %.2f с' % row_major_average)
    print('Среднее время доступа по столбцам: %.2f с' % col_major_average)
    print('Средняя разница по времени:        %.2f с' % (
        (row_major_time-col_major_time) / num_passes))
    print(('Победитель: индексация по ' + winner +
        '; она быстрее на %.2f%%') % pct_better)
```

Выполнение этой программы на ПК с Windows занимает несколько минут. Ниже показаны типичные выходные данные, отображаемые при запуске этой программы.

```
Среднее время доступа по строкам:  16.68 с
Среднее время доступа по столбцам: 15.94 с
Средняя разница по времени:         0.74 с
Победитель: индексация по столбцам; она быстрее на 4.42%
```

Глава 8. Методы повышения производительности

Упражнение 1

Рассмотрим I-кеш L1 с прямым отображением размером 32 Кбайт. Каждая строка кеша состоит из 64 байт, а системное адресное пространство составляет 4 Гбайт. Сколько битов отведено для тега этого кеша? Биты с какими номерами (бит 0 — наименее значимый бит) составляют адресное слово?

Ответ. Кеш содержит 32 768 байт по 64 байта в каждой строке. Количество наборов в кеше: 32 768 : 64 = 512. 512 = 2^9. Таким образом, номер набора имеет длину 9 бит.

Каждая строка кеша содержит 64 (2^6) байта. Это означает, что младшие 6 бит каждого адреса представляют смещение байта в строке кеша.

Для адресного пространства объемом 4 Гбайт требуются 32-разрядные адреса. Вычитание 9 бит номера набора и 6 бит смещения байта из 32-разрядного адреса дает 32 – (9 + 6) = 17 бит в теге кеша.

Тег кеша находится в 17 старших битах адреса, поэтому **эти биты в 32-разрядном адресе занимают область от бита 15 до бита 31**.

Упражнение 2

Рассмотрим 8-канальный наборно-ассоциативный кеш инструкций и данных L2 объемом 256 Кбайт с 64 байтами в каждой строке кеша. Сколько наборов используется в этом кеше?

Ответ. Количество строк в кеше равно 262 144 : 64 = 4096.

Каждый набор содержит 8 строк.

Количество наборов = 4096 строк : 8 строк в наборе = **512 наборов**.

Упражнение 3

Процессор имеет 4-этапный конвейер с максимальными задержками 0,8; 0,4; 0,6 и 0,3 нс на этапах 1–4 соответственно. Если первый этап заменить двумя этапами с максимальными задержками 0,5 и 0,3 нс, соответственно, насколько увеличится тактовая частота процессора в процентном выражении?

Ответ. Максимальная тактовая частота определяется самым медленным этапом конвейера. Самый медленный этап 4-ступенчатого конвейера занимает 0,8 нс. Максимальная тактовая частота равна:

$$1 : (0{,}8 \times 10^{-9}) = 1{,}25 \text{ ГГц}.$$

Самый медленный этап 5-ступенчатого конвейера занимает 0,6 нс. Максимальная тактовая частота равна:

$$1 : (0{,}6 \times 10^{-9}) = 1{,}667 \text{ ГГц}.$$

Увеличение тактовой частоты в результате добавления этапа конвейера составляет:

$$100 \times (1{,}667 \times 10^9 - 1{,}25 \times 10^9) : (1{,}25 \times 10^9) = \mathbf{33{,}3\%}.$$

Глава 9. Специализированные расширения процессоров

Упражнение 1

Используя язык программирования, который предоставляет доступ к байтовому представлению типов данных с плавающей запятой (например, C или C++), напишите функцию, которая принимает в качестве входных данных 32-битное значение одинарной точности. Извлеките знак, порядок и мантиссу из байтов этого значения с плавающей запятой и выведите их на устройство отображения. Удалите смещение из порядка перед выводом его значения и выведите мантиссу в виде десятичного числа. Протестируйте программу на значениях `0`, `-0`, `1`, `-1`, `6.674e-11`, `1.0e38`, `1.0e39`, `1.0e-38` и `1.0e-39`. Перечисленные здесь числовые значения, содержащие *e*, используют текстовое представление чисел с плавающей запятой на языке C/C++. Например, `6.674e-11` означает $6{,}674 \times 10^{-11}$.

Ответ. Файл C++ `Ex__1_float_format.cpp` содержит код для этого упражнения:

```
// Ex__1_float_format.cpp

#include <iostream>
#include <cstdint>

void print_float(float f)
{
    const auto bytes = static_cast<uint8_t*>(static_cast<void*>(&f));

    printf(" Float   | %9g |      ", f);

    for (int i = sizeof(float) - 1; i >= 0; i--)
        printf("%02X", bytes[i]);

    printf("      | ");

    const auto sign = bytes[3] >> 7;
    const auto exponent = ((static_cast<uint16_t>(bytes[3] & 0x7F)
                                        << 8) | bytes[2]) >> 7;
    auto exp_unbiased = exponent - 127;

    uint32_t mantissa = 0;
    for (auto i = 0; i < 3; i++)
        mantissa = (mantissa << 8) | bytes[2 - i];
```

```
    mantissa &= 0x7FFFFF; // Обнулим старший бит

    double mantissa_dec;
    if (exponent == 0)    // Это ноль или денормализованное число
    {
        mantissa_dec = mantissa / static_cast<double>(0x800000);
        exp_unbiased++;
    }
    else
        mantissa_dec = 1.0 + mantissa / static_cast<double>(0x800000);

    printf("  %d  |   %4d   | %lf\n", sign, exp_unbiased, mantissa_dec);
}

int main(void)
{
    printf("  Тип   |   Число   |       Байты       | Знак | Порядок  | Мантисса\n");
    printf(" -------|-----------|-------------------|------|----------|---------\n");

    print_float(0);
    print_float(-0); // Знак "минус" игнорируется
    print_float(1);
    print_float(-1);
    print_float(6.674e-11f);
    print_float(1.0e38f);
    //print_float(1.0e39f); // Ошибка во время компиляции
    print_float(1.0e-38f);
    print_float(1.0e-39f);

    return 0;
}
```

Ниже показаны выходные данные программы:

```
  Тип   |   Число   |       Байты       | Знак | Порядок  | Мантисса
 -------|-----------|-------------------|------|----------|---------
 Float  |         0 |     00000000      |  0   |   -126   | 0.000000
 Float  |         0 |     00000000      |  0   |   -126   | 0.000000
 Float  |         1 |     3F800000      |  0   |      0   | 1.000000
 Float  |        -1 |     BF800000      |  1   |      0   | 1.000000
```

```
Float   | 6.674e-11 |     2E92C348     |  0  |   -34    | 1.146585
Float   |     1e+38 |     7E967699     |  0  |   126    | 1.175494
Float   |     1e-38 |     006CE3EE     |  0  |  -126    | 0.850706
Float   |     1e-39 |     000AE398     |  0  |  -126    | 0.085071
```

К этим результатам есть несколько примечаний.

1. Ноль в IEEE 754 может иметь положительный или отрицательный знак. Нулю, передаваемому в функцию `print_float` во второй строке таблицы, предшествует знак "минус", но этот знак игнорируется при преобразовании в число с плавающей запятой.
2. Значение `1.0e39f` не отображается, поскольку его использование вызывает ошибку во время компиляции: эта константа с плавающей запятой находится вне допустимого диапазона.
3. Ноль представлен в виде мантиссы, равной нулю, и смещенного порядка, равного нулю.
4. Последние две строки содержат числа, которые не могут быть представлены с неявным начальным битом 1 из-за отрицательного переполнения. Эти числа называются **денормализованными** и содержат специальный смещенный порядок относительно 0. Денормализованные числа имеют пониженную точность, т. к. не все биты их мантиссы содержат значимые цифры.
5. Численно денормализованные числа с плавающей запятой фактически используют смещенный показатель, равный 1, который переводится в несмещенный порядок, равный –126.

Упражнение 2

Измените программу из *упражнения 1*, чтобы она могла также принимать значение с плавающей запятой двойной точности и выводить знак, порядок (с удаленным смещением) и мантиссу этого значения. Протестируйте ее с теми же входными значениями, что и в *упражнении 1*, а также со значениями `1.0e308`, `1.0e309`, `1.0e-308` и `1.0e-309`.

Ответ. Файл C++ `Ex__2_double_format.cpp` содержит код для этого упражнения:

```
// Ex__2_double_format.cpp
#include <iostream>
#include <cstdint>

void print_float(float f)
{
    const auto bytes = static_cast<uint8_t*>(static_cast<void*>(&f));
```

```
    printf(" Float  | %9g |      ", f);

    for (int i = sizeof(float) - 1; i >= 0; i--)
        printf("%02X", bytes[i]);

    printf("     | ");

    const auto sign = bytes[3] >> 7;
    const auto exponent = ((static_cast<uint16_t>(bytes[3] & 0x7F)
                                                   << 8) | bytes[2]) >> 7;
    auto exp_unbiased = exponent - 127;

    uint32_t mantissa = 0;
    for (auto i = 0; i < 3; i++)
        mantissa = (mantissa << 8) | bytes[2 - i];

    mantissa &= 0x7FFFFF; // Обнулим старший бит

    double mantissa_dec;
    if (exponent == 0)    // Это ноль или денормализованное число
    {
        mantissa_dec = mantissa / static_cast<double>(0x800000);
        exp_unbiased++;
    }
    else
        mantissa_dec = 1.0 + mantissa / static_cast<double>(0x800000);

    printf("  %d  |   %4d   | %lf\n", sign, exp_unbiased, mantissa_dec);
}

void print_double(double d)
{
    const auto bytes = static_cast<uint8_t*>(static_cast<void*>(&d));

    printf(" Double | %9g | ", d);

    for (int i = sizeof(double) - 1; i >= 0; i--)
        printf("%02X", bytes[i]);

    printf(" | ");
```

```
    const auto sign = bytes[7] >> 7;
    const auto exponent = ((static_cast<uint16_t>(bytes[7] & 0x7F)
                                              << 8) | bytes[6]) >> 4;
    auto exp_unbiased = exponent - 1023;

    uint64_t mantissa = 0;
    for (auto i = 0; i < 7; i++)
        mantissa = (mantissa << 8) | bytes[6 - i];

    mantissa &= 0xFFFFFFFFFFFFF; // Сохраним младшие 52 бита
    double mantissa_dec;
    if (exponent == 0) // Это ноль или денормализованное число
    {
        mantissa_dec = mantissa / static_cast<double>(0x10000000000000);
        exp_unbiased++;
    }
    else
        mantissa_dec = 1.0 + mantissa / static_cast<double>(0x10000000000000);

    printf("  %d  |  %5d   | %lf\n", sign, exp_unbiased, mantissa_dec);
}

int main(void)
{
    printf("  Тип  |    Число    |        Байты       | Знак | Порядок  | Мантисса \n");
    printf(" ------|-----------|------------------|------|----------|----------\n");

    print_float(0);
    print_float(-0);  // Знак "минус" игнорируется
    print_float(1);
    print_float(-1);
    print_float(6.674e-11f);
    print_float(1.0e38f);
    //print_float(1.0e39f); // Ошибка во время компиляции
    print_float(1.0e-38f);
    print_float(1.0e-39f);

    print_double(0);
    print_double(-0); // Знак "минус" игнорируется
    print_double(1);
```

```
    print_double(-1);
    print_double(6.674e-11);
    print_double(1.0e38);
    print_double(1.0e39);
    print_double(1.0e-38);
    print_double(1.0e-39);
    print_double(1.0e308);
    //print_double(1.0e309); // Ошибка во время компиляции
    print_double(1.0e-308);
    print_double(1.0e-309);

    return 0;
}
```

Ниже показаны выходные данные программы:

Тип	Число	Байты	Знак	Порядок	Мантисса
Float	0	00000000	0	-126	0.000000
Float	0	00000000	0	-126	0.000000
Float	1	3F800000	0	0	1.000000
Float	-1	BF800000	1	0	1.000000
Float	6.674e-11	2E92C348	0	-34	1.146585
Float	1e+38	7E967699	0	126	1.175494
Float	1e-38	006CE3EE	0	-126	0.850706
Float	1e-39	000AE398	0	-126	0.085071
Double	0	0000000000000000	0	-1022	0.000000
Double	0	0000000000000000	0	-1022	0.000000
Double	1	3FF0000000000000	0	0	1.000000
Double	-1	BFF0000000000000	1	0	1.000000
Double	6.674e-11	3DD25868F4DEAE16	0	-34	1.146584
Double	1e+38	47D2CED32A16A1B1	0	126	1.175494
Double	1e+39	48078287F49C4A1D	0	129	1.469368
Double	1e-38	380B38FB9DAA78E4	0	-127	1.701412
Double	1e-39	37D5C72FB1552D83	0	-130	1.361129
Double	1e+308	7FE1CCF385EBC8A0	0	1023	1.112537
Double	1e-308	000730D67819E8D2	0	-1022	0.449423
Double	1e-309	0000B8157268FDAF	0	-1022	0.044942

К этим результатам есть несколько примечаний.

1. Ноль в IEEE 754 может иметь положительный или отрицательный знак. Нулю, передаваемому в функцию `print_double` во второй строке таблицы, содержащей тип `Double`, предшествует знак "минус", но этот знак игнорируется при преобразовании в число с плавающей запятой.
2. Значение `1.0e309` не отображается, поскольку его использование вызывает ошибку во время компиляции: "the floating constant is out of range" ("константа с плавающей запятой находится вне допустимого диапазона").
3. Ноль представлен в виде мантиссы, равной нулю, и смещенного порядка, равного нулю.
4. Последние две строки содержат числа, которые не могут быть представлены с неявным начальным битом 1 из-за отрицательного переполнения. Эти числа называются денормализованными и содержат специальный смещенный порядок относительно 0. Денормализованные числа имеют пониженную точность, т. к. не все биты их мантиссы содержат значимые цифры.
5. Численно денормализованные числа с плавающей запятой двойной точности фактически используют смещенный показатель, равный 1, который переводится в несмещенный порядок, равный –1022.

Упражнение 3

Найдите в Интернете информацию о семействе процессоров i.MX RT1060 компании NXP Semiconductors. Скачайте технические описания этого семейства и ответьте на следующие вопросы об этих процессорах.

Ответ. Вводная информация о семействе процессоров i.MX RT1060 доступна по адресу **https://www.nxp.com/docs/en/nxp/data-sheets/IMXRT1060CEC.pdf**.

Полное справочное руководство по i.MX RT1060 доступно только после создания учетной записи по адресу **https://www.nxp.com**.

После входа в свою учетную запись выполните поиск строки `i.MX RT1060 Processor Reference Manual`, чтобы найти справочное руководство и скачать его. Имя файла — `IMXRT1060RM.pdf`.

Упражнение 4

Поддерживают ли процессоры i.MX RT1060 концепцию выполнения инструкций в режиме супервизора? Объясните свой ответ.

Ответ. Поиск по слову `supervisor` в справочном руководстве по процессору i.MX RT1060 дает несколько подсказок. Однако все эти варианты использования относятся к ограничениям доступа, связанным с конкретной подсистемой, такой как модуль FlexCAN.

Режим супервизора в процессоре i.MX RT1060 не работает на уровне выполнения инструкций, т. е. *эти процессоры не реализуют выполнение инструкций в режиме супервизора*, согласно описанному в *главе 9*.

Упражнение 5

Поддерживают ли процессоры i.MX RT1060 концепцию виртуальной памяти со страничной организацией? Объясните свой ответ.

Ответ. Процессоры i.MX RT1060 используют адресацию физической памяти с защитой максимум до 16 областей памяти. *Эти процессоры не поддерживают концепцию виртуальной памяти со страничной организацией.*

Упражнение 6

Поддерживают ли процессоры i.MX RT1060 аппаратные вычисления с плавающей запятой? Объясните свой ответ.

Ответ. В разделе 1.3 "Функции" справочного руководства указана следующая функциональная возможность: **блок вычислений с плавающей запятой одинарной и двойной точности (FPU).**

В "Техническом справочном руководстве по процессору ARM Cortex-M7", которое доступно по адресу **http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B_cortex_m7_trm.pdf**, говорится, что этот FPU обеспечивает поддержку *"вычислений с плавающей запятой, согласно требованиям стандарта ANSI/IEEE 754-2008 — стандарта IEEE для двоичной арифметики с плавающей запятой, называемого также IEEE 754".*

Процессоры i.MX RT1060 поддерживают аппаратные вычисления с плавающей запятой.

Упражнение 7

Какие функции управления питанием поддерживают процессоры i.MX RT1060?

Ответ. В разделе 12.4 справочного руководства описывается подсистема управления питанием процессора. Вот некоторые из ее основных особенностей:

1. Отдельные домены питания для процессора, памяти и остальной части системы.
2. Встроенные вторичные источники питания, обеспечивающие независимую поддержку питания различных подсистем.
3. Регулирование напряжения и тактовой частоты, обеспечивающее **динамическое изменение напряжения и частоты (DVFS).**
4. Датчики температуры.
5. Датчики напряжения.

Упражнение 8

Какие функции безопасности поддерживают процессоры i.MX RT1060?

Ответ. Компоненты безопасности системы описаны в главе 6 справочного руководства. Вот некоторые из основных особенностей:

1. Безопасная загрузка, обеспечивающая проверку цифровой подписи зашифрованного образа кода.
2. Встроенное в чип однократно программируемое ПЗУ для хранения информации, связанной с безопасностью.
3. Аппаратный криптографический сопроцессор, поддерживающий алгоритмы шифрования AES-128, SHA-1 и SHA-256.
4. Генератор истинных случайных чисел для создания безопасных криптографических ключей.
5. Контроллер JTAG для безопасной отладки с поддержкой пароля.
6. Интерфейс с памятью, поддерживающий оперативную расшифровку зашифрованных в ПЗУ данных инструкций.

Глава 10. Современные архитектуры и наборы инструкций процессоров

Упражнение 1

Установите бесплатный выпуск Visual Studio Community, доступный по адресу **https://visualstudio.microsoft.com/vs/community/**, на ПК с ОС Windows. После завершения установки откройте среду разработки Visual Studio IDE и выберите из меню **Tools** (Инструменты) пункт **Get Tools and Features...** (Получить инструменты и возможности...). Установите рабочую нагрузку **Desktop development with C++** (Разработка классических приложений на C++).

В окне поиска Windows на панели задач начните вводить `Developer Command Prompt for VS 2022`. Когда приложение появится в меню поиска, выберите его, чтобы открыть командную строку.

Создайте файл с именем `hello_x86.asm` с содержимым, показанным в листинге исходного кода в *разд. "Язык ассемблера x86" главы 10.*

Выполните сборку программы, используя команду, приведенную в *разд. "Язык ассемблера x86" главы 10*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

Ответ. Установите Visual Studio Community как описано в вопросе, затем установите в Visual Studio Community рабочую нагрузку **Desktop development with C++**:

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__1_hello_x86.asm` содержит следующий пример решения для этого упражнения:

```
.386
.model FLAT,C
.stack 400h

.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near
extern exit:near

public main
main proc
    ; Вывод сообщения
    push    offset message
    call    printf

    ; Выход из программы с кодом состояния 0
    push    0
    call    exit
main endp

.data
message db "Привет, архитектор компьютеров!",0

end
```

2. Откройте **Developer Command Prompt for VS 2022** и перейдите в каталог, содержащий ваш исходный файл.
3. Создайте исполняемый файл с помощью следующей команды:

```
ml /Fl /Zi /Zd Ex__1_hello_x86.asm
```

4. Ниже показаны выходные данные, формируемые программой:

```
C:\>Ex__1_hello_x86.exe
Привет, архитектор компьютеров!
```

Это файл листинга, созданный процедурой сборки:

```
Microsoft (R) Macro Assembler Version 14.31.31104.0      02/21/22 07:39:20
Ex__1_hello_x86.asm                                       Page 1 - 1

        .386
        .model FLAT,C
        .stack 400h

 00000000     .code
        includelib libcmt.lib
        includelib legacy_stdio_definitions.lib

        extern printf:near
        extern exit:near

        public main
 00000000     main proc
            ; Вывод сообщения
 00000000  68 00000000 R      push    offset message
 00000005  E8 00000000 E      call    printf

            ; Выход из программы с кодом состояния 0
 0000000A  6A 00              push    0
 0000000C  E8 00000000 E      call    exit
 00000011     main endp

 00000000     .data
 00000000 48 65 6C 6C 6F    message db "Привет, архитектор компьютеров!",0
          2C 20 43 6F 6D
          70 75 74 65 72
          20 41 72 63 68
          69 74 65 63 74
          21 00

        end
```

```
Microsoft (R) Macro Assembler Version 14.31.31104.0        02/21/22 07:39:20
Ex__1_hello_x86.asm                                        Symbols 2 - 1

Segments and Groups:

           N a m e                 Size     Length   Align    Combine Class

FLAT . . . . . . . . . . . . . .   GROUP
STACK  . . . . . . . . . . . . .   32 Bit   00000400 DWord    Stack
'STACK'
_DATA  . . . . . . . . . . . . .   32 Bit   0000001B DWord    Public  'DATA'
_TEXT  . . . . . . . . . . . . .   32 Bit   00000011 DWord    Public  'CODE'

Procedures, parameters, and locals:

                N a m e                Type     Value    Attr

main . . . . . . . . . . . . . .   P Near   00000000 _TEXT Length= 00000011


Symbols:

                N a m e                Type     Value    Attr

@CodeSize  . . . . . . . . . . .   Number   00000000h
@DataSize  . . . . . . . . . . .   Number   00000000h
@Interface . . . . . . . . . . .   Number   00000001h
@Model . . . . . . . . . . . . .   Number   00000007h
@code  . . . . . . . . . . . . .   Text     _TEXT
@data  . . . . . . . . . . . . .   Text     FLAT
@fardata?  . . . . . . . . . . .   Text     FLAT
@fardata . . . . . . . . . . . .   Text     FLAT
@stack . . . . . . . . . . . . .   Text     FLAT
exit . . . . . . . . . . . . . .   L Near   00000000 FLAT   External C
message  . . . . . . . . . . . .   Byte     00000000 _DATA
printf . . . . . . . . . . . . .   L Near   00000000 FLAT   External C

     0 Warnings
     0 Errors
```

Упражнение 2

Напишите на ассемблере *x*86 программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – 66) × (445 + 136)] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__2_expr_x86.asm` содержит следующий пример решения для этого упражнения:

```
.386
.model FLAT,C
.stack 400h

.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near
extern exit:near

public main
main proc
    ; Напечатаем начальную строку вывода
    push    offset msg1
    call    printf

    ; Вычислим [(129 - 66) * (445 + 136)] / 3
    mov     eax, 129
    sub     eax, 66
    mov     ebx, 445
    add     ebx, 136
    mul     bx
    mov     bx, 3
    div     bx

    ; Напечатаем старший байт
    push    eax
    mov     bl, ah
    call    print_byte
```

```
    ; Напечатаем младший байт
    pop     ebx
    call    print_byte

    ; Напечатаем завершающую строку вывода
    push    offset msg2
    call    printf

    push    0
    call    exit
main endp

; Передадим байт для печати в ebx
print_byte proc
    ; Пролог функции x86
    push    ebp
    mov     ebp, esp

    ; Используем функцию printf библиотеки C
    and     ebx, 0ffh
    push    ebx
    push    offset fmt_str
    call    printf

    ; Эпилог функции x86
    mov     esp, ebp
    pop     ebp
    ret
print_byte endp

.data
fmt_str db "%02X", 0
msg1    db "[(129 - 66) * (445 + 136)] / 3 = ", 0
msg2    db "h", 9

end
```

2. Откройте **Developer Command Prompt for VS 2022** и перейдите в каталог, содержащий ваш исходный файл.

3. Создайте исполняемый файл с помощью следующей команды:

```
ml /Fl /Zi /Zd Ex__1_hello_x86.asm
```

4. Ниже показаны выходные данные, формируемые программой:

```
C:\>Ex__2_expr_x86.exe
[(129 - 66) * (445 + 136)] / 3 = 2FA9h
```

Это файл листинга, созданный процедурой сборки:

```
Microsoft (R) Macro Assembler Version 14.31.31104.0      02/21/22 07:42:23
Ex__2_expr_x86.asm                                       Page 1 - 1


                                   .386
                                   .model FLAT,C
                                   .stack 400h

 00000000                          .code
                                    includelib libcmt.lib
                                    includelib legacy_stdio_definitions.lib

                                    extern printf:near
                                    extern exit:near

                                    public main
 00000000                           main proc
                                       ; Напечатаем начальную строку вывода
 00000000  68 00000005 R               push    offset msg1
 00000005  E8 00000000 E               call    printf

                                       ; Вычислим [(129 - 66) * (445 + 136)] / 3
 0000000A  B8 00000081                 mov     eax, 129
 0000000F  83 E8 42                    sub     eax, 66
 00000012  BB 000001BD                 mov     ebx, 445
 00000017  81 C3 00000088              add     ebx, 136
 0000001D  66| F7 E3                   mul     bx
 00000020  66| BB 0003                 mov     bx, 3
 00000024  66| F7 F3                   div     bx
```

```
                                ; Напечатаем старший байт
00000027  50                    push    eax
00000028  8A DC                 mov     bl, ah
0000002A  E8 00000017           call    print_byte

                                ; Напечатаем младший байт
0000002F  5B                    pop     ebx
00000030  E8 00000011           call    print_byte

                                ; Напечатаем завершающую строку вывода
00000035  68 00000027 R         push    offset msg2
0000003A  E8 00000000 E         call    printf

0000003F  6A 00                 push    0
00000041  E8 00000000 E         call    exit
00000046                    main endp

                            ; Передадим байт для печати в ebx
00000046                    print_byte proc
                            ; Пролог функции x86
00000046  55                push    ebp
00000047  8B EC             mov     ebp, esp

                                ; Используем функцию printf библиотеки C
00000049  81 E3 000000FF        and     ebx, 0ffh
0000004F  53                    push    ebx
00000050  68 00000000 R         push    offset fmt_str
00000055  E8 00000000 E         call    printf

                                ; Эпилог функции x86
0000005A  8B E5                 mov     esp, ebp

0000005C  5D                    pop     ebp
0000005D  C3                    ret
0000005E                    print_byte endp

00000000                    .data
00000000 25 30 32 58 00     fmt_str db "%02X", 0
```

```
 00000005 5B 28 31 32 39     msg1    db "[(129 - 66) * (445 + 136)] / 3 = ", 0
          20 2D 20 36 36
          29 20 2A 20 28
          34 34 35 20 2B
          20 31 33 36 29
          5D 20 2F 20 33
             20 3D 20 00
 00000027 68 09              msg2    db "h", 9

                             end
Microsoft (R) Macro Assembler Version 14.31.31104.0       02/21/22 07:42:23
Ex__2_expr_x86.asm                                        Symbols 2 - 1



Segments and Groups:

                N a m e                 Size     Length   Align   Combine Class

FLAT . . . . . . . . . . . . . .     GROUP
STACK  . . . . . . . . . . . . .     32 Bit   00000400 DWord    Stack    'STACK'
_DATA  . . . . . . . . . . . . .     32 Bit   00000029 DWord    Public   'DATA'
_TEXT  . . . . . . . . . . . . .     32 Bit   0000005E DWord    Public   'CODE'

Procedures, parameters, and locals:

                N a m e                 Type     Value    Attr

main . . . . . . . . . . . . . . . P Near    00000000 _TEXT  Length= 00000046 Public C
print_byte . . . . . . . . . . . . P Near    00000046 _TEXT  Length= 00000018 Public C


Symbols:

                N a m e                 Type       Value    Attr

@CodeSize  . . . . . . . . . . . .   Number     00000000h
@DataSize  . . . . . . . . . . . .   Number     00000000h
@Interface . . . . . . . . . . . .   Number     00000001h
```

```
@Model . . . . . . . . . . . . . .  Number   00000007h
@code  . . . . . . . . . . . . . .  Text     _TEXT
@data  . . . . . . . . . . . . . .  Text     FLAT
@fardata?  . . . . . . . . . . . .  Text     FLAT
@fardata . . . . . . . . . . . . .  Text     FLAT
@stack . . . . . . . . . . . . . .  Text     FLAT
exit . . . . . . . . . . . . . . .  L Near   00000000 FLAT   External C
fmt_str  . . . . . . . . . . . . .  Byte     00000000 _DATA
msg1 . . . . . . . . . . . . . . .  Byte     00000005 _DATA
msg2 . . . . . . . . . . . . . . .  Byte     00000027 _DATA
printf . . . . . . . . . . . . . .  L Near   00000000 FLAT   External C

     0 Warnings
     0 Errors
```

Упражнение 3

В окне поиска Windows на панели задач начните вводить `x64 Native Tools Command Prompt for VS 2022`. Когда приложение появится в меню поиска, выберите его, чтобы открыть командную строку:

Создайте файл с именем `hello_x64.asm` с содержимым, показанным в листинге исходного кода в *разд. "Язык ассемблера x64" главы 10*.

Выполните сборку программы, используя команду, приведенную в *разд. "Язык ассемблера x64" главы 10*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__3_hello_x64.asm` содержит следующий пример решения для этого упражнения:

```
.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near
extern exit:near

public main
main proc
```

```
    ; Резервирование места в стеке
    sub     rsp, 40

    ; Вывод сообщения
    lea     rcx, message
    call    printf

    ; Выход из программы с кодом состояния 0
    xor     rcx, rcx
    call    exit
main endp

.data
message db "Привет, архитектор компьютеров!",0

end
```

2. Откройте **x64 Native Tools Command Prompt for VS 2019** и перейдите в каталог, содержащий ваш исходный файл.
3. Создайте исполняемый файл с помощью этой команды:

```
ml64 /Fl /Zi /Zd Ex__3_hello_x64.asm
```

4. Ниже показаны выходные данные, формируемые программой:

```
C:\>Ex__3_hello_x64.exe
Привет, архитектор компьютеров!
```

Это файл листинга, созданный процедурой сборки:

```
Microsoft (R) Macro Assembler (x64) Version 14.31.31104.0   02/21/22 07:47:41
Ex__3_hello_x64.asm                                          Page 1 - 1

 00000000                      .code
                               includelib libcmt.lib
                               includelib legacy_stdio_definitions.lib

                               extern printf:near
                               extern exit:near

                               public main
 00000000                      main proc
```

```
                                ; Резервирование места в стеке
 00000000  48/ 83 EC 28         sub     rsp, 40

                                ; Вывод сообщения
 00000004  48/ 8D 0D            lea     rcx, message
        00000000 R
 0000000B  E8 00000000 E        call    printf

                                ; Выход из программы с кодом состояния 0
 00000010  48/ 33 C9            xor     rcx, rcx
 00000013  E8 00000000 E        call    exit
 00000018                       main endp

 00000000                   .data
 00000000 48 65 6C 6C 6F    message db "Привет, архитектор компьютеров!",0
          2C 20 43 6F 6D
          70 75 74 65 72
          20 41 72 63 68
          69 74 65 63 74
          21 00

                            end
Microsoft (R) Macro Assembler (x64) Version 14.31.31104.0   02/21/22 07:47:41
Ex__3_hello_x64.asm                                         Symbols 2 - 1

Procedures, parameters, and locals:

                N a m e                 Type     Value     Attr

main . . . . . . . . . . . . . .  P    00000000 _TEXT  Length= 00000018 Public

Symbols:
                N a m e                Type     Value    Attr

exit . . . . . . . . . . . . . .   L        00000000 _TEXT   External
message  . . . . . . . . . . . .   Byte     00000000 _DATA
printf . . . . . . . . . . . . .   L        00000000 _TEXT   External

    0 Warnings
    0 Errors
```

Упражнение 4

Напишите на ассемблере *x*64 программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – 66) × (445 + 136)] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__4_expr_x64.asm` содержит следующий пример решения для этого упражнения:

```
.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near
extern exit:near

public main
main proc
    ; Резервирование места в стеке
    sub     rsp, 40

    ; Напечатаем начальную строку вывода
    lea     rcx, msg1
    call    printf

    ; Вычислим [(129 - 66) * (445 + 136)] / 3
    mov     eax, 129
    sub     eax, 66
    mov     ebx, 445
    add     ebx, 136
    mul     bx
    mov     bx, 3
    div     bx

    ; Напечатаем старший байт
    push    rax
    mov     bl, ah
    and     ebx, 0ffh
    call    print_byte
```

```
    ; Напечатаем младший байт
    pop     rbx
    and     ebx, 0ffh
    call    print_byte

    ; Напечатаем завершающую строку вывода
    lea     rcx, msg2
    call    printf

    ; Выход из программы с кодом состояния 0
    xor     rcx, rcx
    call    exit
main endp

; Передадим байт для печати в ebx
print_byte proc
    ; Пролог функции x64
    sub     rsp, 40

    ; Используем функцию printf библиотеки C
    mov     rdx, rbx
    lea     rcx, fmt_str
    call    printf

    ; Эпилог функции x64
    add     rsp, 40

    ret
print_byte endp

.data
fmt_str db "%02X", 0
msg1    db "[(129 - 66) * (445 + 136)] / 3 = ", 0
msg2    db "h", 9
end
```

2. Откройте **x64 Native Tools Command Prompt for VS 2019** и перейдите в каталог, содержащий ваш исходный файл.

3. Создайте исполняемый файл с помощью этой команды:

```
ml64 /Fl /Zi /Zd Ex__3_hello_x64.asm
```

4. Ниже показаны выходные данные, формируемые программой:

```
C:\>Ex__4_expr_x64.exe
[(129 - 66) * (445 + 136)] / 3 = 2FA9h
```

Это файл листинга, созданный процедурой сборки:

```
Microsoft (R) Macro Assembler (x64) Version 14.31.31104.0   02/21/22 07:49:37
Ex__4_expr_x64.asm                                          Page 1 - 1


 00000000                     .code
                              includelib libcmt.lib
                              includelib legacy_stdio_definitions.lib

                              extern printf:near
                              extern exit:near

                              public main
 00000000                     main proc
                                  ; Зарезервируем место в стеке
 00000000  48/ 83 EC 28           sub     rsp, 40

                                  ; Напечатаем начальную строку вывода
 00000004  48/ 8D 0D              lea     rcx, msg1
        00000005 R
 0000000B  E8 00000000 E          call    printf

                                  ; Вычислим [(129 - 66) * (445 + 136)] / 3
 00000010  B8 00000081            mov     eax, 129
 00000015  83 E8 42               sub     eax, 66
 00000018  BB 000001BD            mov     ebx, 445
 0000001D  81 C3 00000088         add     ebx, 136
 00000023  66| F7 E3              mul     bx
 00000026  66| BB 0003            mov     bx, 3
 0000002A  66| F7 F3              div     bx
```

```
                                      ; Напечатаем старший байт
0000002D  50                          push    rax
0000002E  8A DC                       mov     bl, ah
00000030  81 E3 000000FF              and     ebx, 0ffh
00000036  E8 00000020                 call    print_byte

                                      ; Напечатаем младший байт
0000003B  5B                          pop     rbx
0000003C  81 E3 000000FF              and     ebx, 0ffh
00000042  E8 00000014                 call    print_byte

                                      ; Напечатаем завершающую строку вывода
00000047  48/ 8D 0D                   lea     rcx, msg2
       00000027 R
0000004E  E8 00000000 E               call    printf

                                      ; Выход из программы с кодом состояния 0
00000053  48/ 33 C9                   xor     rcx, rcx
00000056  E8 00000000 E               call    exit
0000005B                          main endp

                                  ; Передадим байт для печати в ebx
0000005B                          print_byte proc
                                  ; Пролог функции x64
0000005B  48/ 83 EC 28                sub     rsp, 40

                                      ; Используем функцию printf библиотеки C
0000005F  48/ 8B D3                   mov     rdx, rbx
00000062  48/ 8D 0D                   lea     rcx, fmt_str
       00000000 R

00000069  E8 00000000 E               call    printf

                                  ; Эпилог функции x6
0000006E  48/ 83 C4 28                add     rsp, 40

00000072  C3                      ret
00000073                          print_byte endp
```

```
 00000000                    .data
 00000000 25 30 32 58 00     fmt_str db "%02X", 0
 00000005 5B 28 31 32 39     msg1    db "[(129 - 66) * (445 + 136)] / 3 = ", 0
          20 2D 20 36 36
          29 20 2A 20 28
          34 34 35 20 2B
          20 31 33 36 29
          5D 20 2F 20 33
             20 3D 20 00
 00000027 68 09                      msg2    db "h", 9

                             end
Microsoft (R) Macro Assembler (x64) Version 14.31.31104.0   02/21/22 07:49:37
Ex__4_expr_x64.asm                                           Symbols 2 - 1




Procedures, parameters, and locals:

                N a m e                 Type     Value    Attr

main . . . . . . . . . . . . . .    P     00000000 _TEXT Length= 0000005B Public
print_byte . . . . . . . . . . .    P     0000005B _TEXT Length= 00000018 Public


Symbols:

                N a m e                    Type     Value     Attr

exit . . . . . . . . . . . . . .     L        00000000 _TEXT  External
fmt_str  . . . . . . . . . . . .     Byte     00000000 _DATA
msg1 . . . . . . . . . . . . . .     Byte     00000005 _DATA
msg2 . . . . . . . . . . . . . .     Byte     00000027 _DATA
printf . . . . . . . . . . . . .     L        00000000 _TEXT  External

     0 Warnings
     0 Errors
```

Упражнение 5

Установите бесплатный пакет Android Studio IDE, доступный по адресу **https://developer.android.com/studio/**. После завершения установки откройте среду разработки Android Studio IDE и выберите из меню **Tools** (Инструменты) пункт **SDK Manager** (Диспетчер SDK). Выберите вкладку **SDK Tools** (Инструменты SDK) и установите флажок **NDK**, который может иметь обозначение **NDK (Side by side)**. Завершите установку NDK (**NDK** расшифровывается как **собственный пакет разработки**).

Найдите следующие файлы в каталоге установки SDK (местоположение по умолчанию: `%LOCALAPPDATA%\Android`) и добавьте эти каталоги в переменную среды `PATH`: `arm-linux-androideabi-as.exe` и `adb.exe`. Подсказка: следующая команда предназначена для конкретной версии Android Studio (ваш путь может отличаться):

```
set PATH=%PATH%;%LOCALAPPDATA%\Android\ Sdk\ndk\23.0.7599858\
toolchains\llvm\prebuilt\windows-x86_64\bin;%LOCALAPPDATA%\Android\
Sdk\platform-tools
```

Создайте файл с именем `hello_arm.s` с содержимым, показанным в листинге исходного кода в *разд. "32-разрядный язык ассемблера ARM" главы 10.*

Выполните сборку программы, используя команду, приведенную в *разд. "32-разрядный язык ассемблера ARM" главы 10.*

Включите **режим разработчика** на телефоне или планшете Android. Указания о том, как это сделать, можно найти в Интернете.

Подсоедините Android-устройство к компьютеру с помощью USB-кабеля.

Скопируйте исполняемый файл программы на телефон, используя команды, приведенные в *разд. "32-разрядный язык ассемблера ARM" главы 10*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__5_hello_arm.s` содержит следующий пример решения для этого упражнения:

```
.text
.global _start

_start:
    // Вывод сообщения в файл 1 (stdout) с помощью системного вызова 4
    mov     r0, #1
    ldr     r1, =msg
    mov     r2, #msg_len
    mov     r7, #4
    svc     0
```

```
    // Выход из программы с помощью системного вызова 1 с возвращением состояния 0
    mov     r0, #0
    mov     r7, #1
    svc     0

.data
msg:
    .ascii      "Привет, архитектор компьютеров!"
msg_len = . - msg
```

2. Создайте исполняемый файл с помощью этих команд:

```
arm-linux-androideabi-as -al=Ex__5_hello_arm.lst -o Ex__5_hello_arm.o
Ex__5_hello_arm.s
arm-linux-androideabi-ld -o Ex__5_hello_arm Ex__5_hello_arm.o
```

3. Следующие строки отображаются после копирования этой программы на устройство Android и ее запуска:

```
C:\>adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
9826f541374f4b4a68 device
C:\>adb push Ex__5_hello_arm /data/local/tmp/Ex__5_hello_arm
Ex__5_hello_arm: 1 file pushed. 0.0 MB/s (868 bytes in 0.059s)
C:\>adb shell chmod +x /data/local/tmp/Ex__5_hello_arm
C:\>adb shell /data/local/tmp/Ex__5_hello_arm
Привет, архитектор компьютеров!
```

Это файл листинга, созданный процедурой сборки:

```
ARM GAS   Ex__5_hello_arm.s                 page 1


   1                .text
   2                .global _start
   3
   4                _start:
   5                    // Вывод сообщения в файл 1 (stdout) с помощью системного
вызова 4
```

```
  6 0000 0100A0E3      mov      r0, #1
  7 0004 14109FE5      ldr      r1, =msg
  8 0008 1A20A0E3      mov      r2, #msg_len
  9 000c 0470A0E3      mov      r7, #4
 10 0010 000000EF      svc      0
 11
 12                    // Выход из программы с помощью системного вызова 1 с
возвращением состояния 0
 13 0014 0000A0E3      mov      r0, #0
 14 0018 0170A0E3      mov      r7, #1
 15 001c 000000EF      svc      0
 16
 17                 .data
 18                 msg:
 19 0000 48656C6C      .ascii      "Привет, архитектор компьютеров!"
 19      6F2C2043
 19      6F6D7075
 19      74657220
 19      41726368
 20                 msg_len = . - msg
```

Упражнение 6

Напишите на 32-разрядном ассемблере ARM программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: $[(129 - 66) \times (445 + 136)] : 3$. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__6_expr_arm.s` содержит следующий пример решения для этого упражнения:

```
.text
.global _start

_start:
    // Напечатаем начальную строку вывода
    ldr     r1, =msg1
    mov     r2, #msg1_len
    bl      print_string
```

```
    // Вычислим [(129 - 66) * (445 + 136)] / 3
    mov     r0, #129
    sub     r0, r0, #66
    ldr     r1, =#445
    add     r1, r1, #136
    mul     r0, r1, r0
    mov     r1, #3
    udiv    r0, r0, r1

    // Напечатаем старший байт результата
    push    {r0}
    lsr     r0, r0, #8
    bl      print_byte

    // Напечатаем младший байт результата
    pop     {r0}
    bl      print_byte

    // Напечатаем завершающую строку вывода
    ldr     r1, =msg2
    mov     r2, #msg2_len
    bl      print_string

    // Выход из программы с помощью системного вызова 1 с возвращением состояния
0
    mov     r0, #0
    mov     r7, #1
    svc     0

// Напечатаем строку; r1=адрес строки, r2=длина строки:
    mov     r0, #1
    mov     r7, #4
    svc     0
    mov     pc, lr

// Преобразуем младшие 4 бита r0 в символ ascii в r0
nibble2ascii:
    and     r0, #0xF
    cmp     r0, #10
```

```
    addpl   r0, r0, #('A' - 10)
    addmi   r0, r0, #'0'
    mov     pc, lr

// Напечатаем байт в шестнадцатеричном формате
print_byte:
    push    {lr}
    push    {r0}
    lsr     r0, r0, #4
    bl      nibble2ascii
    ldr     r1, =bytes
    strb    r0, [r1], #1

    pop     {r0}
    bl      nibble2ascii
    strb    r0, [r1]
    ldr     r1, =bytes
    mov     r2, #2
    bl      print_string

    pop     {lr}
    mov     pc, lr

.data
msg1:
    .ascii  "[(129 - 66) * (445 + 136)] / 3 = "
msg1_len = . - msg1

bytes:
    .ascii  "??"

msg2:
    .ascii  "h"
msg2_len = . - msg2
```

2. Создайте исполняемый файл с помощью этих команд:

```
arm-linux-androideabi-as -al=Ex__6_expr_arm.lst -o Ex__6_expr_arm.o
Ex__6_expr_arm.s
arm-linux-androideabi-ld -o Ex__6_expr_arm Ex__6_expr_arm.o
```

3. Следующие строки отображаются после копирования этой программы на устройство Android и ее запуска:

```
C:\>adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
9826f541374f4b4a68      device


C:\>adb push Ex__6_expr_arm /data/local/tmp/Ex__6_expr_arm
Ex__6_expr_arm: 1 file pushed. 0.2 MB/s (1188 bytes in 0.007s)

C:\>adb shell chmod +x /data/local/tmp/Ex__6_expr_arm
C:\>adb shell /data/local/tmp/Ex__6_expr_arm
[(129 - 66) * (445 + 136)] / 3 = 2FA9h
```

Это файл листинга, созданный процедурой сборки:

```
                        .text
                        .global _start

                        _start:
                            // Напечатаем начальную строку вывода
  0000 A8109FE5             ldr     r1, =msg1
  0004 2120A0E3             mov     r2, #msg1_len
  0008 110000EB             bl      print_string

                            // Вычислим [(129 - 66) * (445 + 136)] / 3
  000c 8100A0E3             mov     r0, #129
  0010 420040E2             sub     r0, r0, #66
  0014 98109FE5             ldr     r1, =#445
  0018 881081E2             add     r1, r1, #136
  001c 910000E0             mul     r0, r1, r0
  0020 0310A0E3             mov     r1, #3
  0024 10F130E7             udiv    r0, r0, r1

                            // Напечатаем старший байт результата
  0028 04002DE5             push    {r0}
  002c 2004A0E1             lsr     r0, r0, #8
```

```
0030 100000EB       bl      print_byte

                    // Напечатаем младший байт результата
0034 04009DE4       pop     {r0}
0038 0E0000EB       bl      print_byte

                    // Напечатаем завершающую строку вывода
003c 74109FE5       ldr     r1, =msg2
0040 0120A0E3       mov     r2, #msg2_len
0044 020000EB       bl      print_string

                // Выход из программы с помощью системного вызова 1
с возвращением состояния 0

0048 0000A0E3       mov     r0, #0
004c 0170A0E3       mov     r7, #1
0050 000000EF       svc     0

                // Напечатаем строку; r1=адрес строки, r2=длина строки
                print_string:
0054 0100A0E3       mov     r0, #1
0058 0470A0E3       mov     r7, #4
005c 000000EF       svc     0
0060 0EF0A0E1       mov     pc, lr

                // Преобразуем младшие 4 бита r0 в символ ascii в r0
                nibble2ascii:
0064 0F0000E2       and     r0, #0xF
0068 0A0050E3       cmp     r0, #10
006c 37008052       addpl   r0, r0, #('A' - 10)
0070 30008042       addmi   r0, r0, #'0'
0074 0EF0A0E1       mov     pc, lr

                // Напечатаем байт в шестнадцатеричном формате
                print_byte:
0078 04E02DE5       push    {lr}
007c 04002DE5       push    {r0}
0080 2002A0E1       lsr     r0, r0, #4
```

```
ARM GAS   Ex__6_expr_arm.s                                          page 2


  58 0084 F6FFFFEB        bl      nibble2ascii
  59 0088 2C109FE5        ldr     r1, =bytes
  60 008c 0100C1E4        strb    r0, [r1], #1
  61
  62 0090 04009DE4        pop     {r0}
  63 0094 F2FFFFEB        bl      nibble2ascii
  64 0098 0000C1E5        strb    r0, [r1]
  65
  66 009c 18109FE5        ldr     r1, =bytes
  67 00a0 0220A0E3        mov     r2, #2
  68 00a4 EAFFFFEB        bl      print_string
  69
  70 00a8 04E09DE4        pop     {lr}
  71 00ac 0EF0A0E1        mov     pc, lr
  72
  73                  .data
  74                  msg1:
  75 0000 5B283132        .ascii  "[(129 - 66) * (445 + 136)] / 3 = "
  75      39202D20
  75      36362920
  75      2A202834
  75      3435202B
  76                  msg1_len = . - msg1
  77
  78                  bytes:
  79 0021 3F3F            .ascii  "??"
  80
  81                  msg2:
  82 0023 68              .ascii  "h"
  83                  msg2_len = . - msg2
```

Упражнение 7

Найдите следующие файлы в каталоге установки Android SDK (местоположение по умолчанию: `%LOCALAPPDATA%\Android`) и добавьте эти каталоги в переменную среды `PATH`: `aarch64- linux-android-as.exe` и `adb.exe`.

Подсказка: следующая команда предназначена для конкретной версии Android Studio (ваш путь может отличаться):

```
set
PATH=%PATH%;%LOCALAPPDATA%\Android\Sdk\ndk\23.0.7599858\toolchains\llvm\prebuilt\w
indows-x86_64\bin;%LOCALAPPDATA%\Android\Sdk\platform-tools
```

Создайте файл с именем `hello_arm64.s` с содержимым, показанным в листинге исходного кода в *разд. "64-разрядный язык ассемблера ARM" главы 10.*

Выполните сборку программы, используя команду, приведенную в *разд. "64-разрядный язык ассемблера ARM" главы 10.*

Включите **режим разработчика** на телефоне или планшете Android.

Подсоедините Android-устройство к компьютеру с помощью USB-кабеля.

Скопируйте исполняемый файл программы на телефон, используя команды, приведенные в *разд. "64-разрядный язык ассемблера ARM" главы 10*, и запустите ее. Убедитесь, что на экране отображается строка: "Привет, архитектор компьютеров!".

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__7_hello_arm64.s` содержит следующий пример решения для этого упражнения:

```
.text
.global _start

_start:
    // Вывод сообщения в файл 1 (stdout) с помощью системного вызова 64
    mov     x0, #1
    ldr     x1, =msg
    mov     x2, #msg_len
    mov     x8, #64
    svc     0

    // Выход из программы с помощью системного вызова 93 с возвращением состояния
0
    mov     x0, #0
    mov     x8, #93
    svc     0

.data
msg:
    .ascii      "Привет, архитектор компьютеров!"
msg_len = . - msg
```

2. Создайте исполняемый файл с помощью этих команд:

```
aarch64-linux-android-as -al=Ex__7_hello_arm64.lst -o Ex__7_hello_arm64.o
Ex__7_hello_arm64.s
aarch64-linux-android-ld -o Ex__7_hello_arm64 Ex__7_hello_arm64.o
```

3. Следующие строки отображаются после копирования этой программы на устройство Android и ее запуска:

```
C:\>adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
9826f541374f4b4a68      device


C:\>adb push Ex__7_hello_arm64 /data/local/tmp/Ex__7_hello_arm64
Ex__7_hello_arm64: 1 file pushed. 0.0 MB/s (1152 bytes in 0.029s)

C:\>adb shell chmod +x /data/local/tmp/Ex__7_hello_arm64
C:\>adb shell /data/local/tmp/Ex__7_hello_arm64
Привет, архитектор компьютеров!
```

Это файл листинга, созданный процедурой сборки:

```
AARCH64 GAS  Ex__7_hello_arm64.s                                page 1


   1                    .text
   2                    .global _start
   3
   4                    _start:
   5                        // Вывод сообщения в файл 1 (stdout) с помощью системного
вызова 64
   6 0000 200080D2          mov     x0, #1
   7 0004 E1000058          ldr     x1, =msg
   8 0008 420380D2          mov     x2, #msg_len
   9 000c 080880D2          mov     x8, #64
  10 0010 010000D4          svc     0
  11
  12                        // Выход из программы с помощью системного вызова 93
с возвращением состояния 0
```

```
13 0014 000080D2        mov     x0, #0
14 0018 A80B80D2        mov     x8, #93
15 001c 010000D4        svc     0
16
17                  .data
18                  msg:
19 0000 48656C6C        .ascii      "Привет, архитектор компьютеров!"
19      6F2C2043
19      6F6D7075
19      74657220
19      41726368
20                  msg_len = . - msg
```

Упражнение 8

Напишите на 64-разрядном ассемблере ARM программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: $[(129 - 66) \times (445 + 136)] : 3$. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

Ответ

1. Создайте свой исходный файл на языке ассемблера. Файл `Ex__8_expr_arm64.s` содержит следующий пример решения для этого упражнения:

```
.text
.global _start

_start:
    // Напечатаем начальную строку вывода
    ldr     x1, =msg1
    mov     x2, #msg1_len
    bl      print_string

    // Вычислим [(129 - 66) * (445 + 136)] / 3
    mov     x0, #129
    sub     x0, x0, #66
    mov     x1, #445
    add     x1, x1, #136
    mul     x0, x1, x0
```

```
    mov     x1, #3
    udiv    x0, x0, x1

    // Напечатаем старший байт результата
    mov     x19, x0
    lsr     x0, x0, #8
    bl      print_byte

    // Напечатаем младший байт результата
    mov     x0, x19
    bl      print_byte

    // Напечатаем завершающую строку вывода
    ldr     x1, =msg2
    mov     x2, #msg2_len
    bl      print_string

    // Выход из программы с помощью системного вызова 93 с возвращением состояния 0
    mov     x0, #0
    mov     x8, #93
    svc     0

// Напечатаем строку; x1=адрес строки, x2=длина строки
print_string:
    mov     x0, #1
    mov     x8, #64
    svc     0
    ret     x30

// Преобразуем младшие 4 бита x0 в символ ascii в x0
nibble2ascii:
    and     x0, x0, #0xF
    cmp     x0, #10
    bmi     lt10

    add     x0, x0, #('A' - 10)
    b       done

lt10:
    add     x0, x0, #'0'
```

```
done:
    ret     x30

// Напечатаем байт в шестнадцатеричном формате
print_byte:
    mov     x21, x30
    mov     x20, x0
    lsr     x0, x0, #4
    bl      nibble2ascii
    ldr     x1, =bytes
    strb    w0, [x1], #1

    mov     x0, x20
    bl      nibble2ascii
    strb    w0, [x1]

    ldr     x1, =bytes
    mov     x2, #2
    bl      print_string

    mov     x30, x21
    ret     x30

.data
msg1:
    .ascii  "[(129 - 66) * (445 + 136)] / 3 = "
msg1_len = . - msg1

bytes:
    .ascii  "??"

msg2:
    .ascii  "h"
msg2_len = . - msg2
```

2. Создайте исполняемый файл с помощью этих команд:

```
aarch64-linux-android-as -al=Ex__8_expr_arm64.lst -o Ex__8_expr_arm64.o
Ex__8_expr_arm64.s
aarch64-linux-android-ld -o Ex__8_expr_arm64 Ex__8_expr_arm64.o
```

3. Следующие строки отображаются после копирования этой программы на устройство Android и ее запуска:

```
C:\>adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
9826f541374f4b4a68      device


C:\>adb push Ex__8_expr_arm64 /data/local/tmp/Ex__8_expr_arm64
Ex__8_expr_arm64: 1 file pushed. 0.1 MB/s (1592 bytes in 0.015s)

C:\>adb shell chmod +x /data/local/tmp/Ex__8_expr_arm64
C:\>adb shell /data/local/tmp/Ex__8_expr_arm64
[(129 - 66) * (445 + 136)] / 3 = 2FA9h
```

Это файл листинга, созданный процедурой сборки:

```
                      .text
                      .global _start

                      _start:
                         // Напечатаем начальную строку вывода
  0000 C1050058          ldr     x1, =msg1
  0004 220480D2          mov     x2, #msg1_len
  0008 13000094          bl      print_string

                         // Вычислим [(129 - 66) * (445 + 136)] / 3
  000c 201080D2          mov     x0, #129
  0010 000801D1          sub     x0, x0, #66
  0014 A13780D2          mov     x1, #445
  0018 21200291          add     x1, x1, #136
  001c 207C009B          mul     x0, x1, x0
  0020 610080D2          mov     x1, #3
  0024 0008C19A          udiv    x0, x0, x1

                         // Напечатаем старший байт результата
  0028 F30300AA          mov     x19, x0
  002c 00FC48D3          lsr     x0, x0, #8
```

```
 0030 14000094        bl      print_byte

                      // Напечатаем младший байт результата
 0034 E00313AA        mov     x0, x19
 0038 12000094        bl      print_byte

                      // Напечатаем завершающую строку вывода
 003c 21040058        ldr     x1, =msg2
 0040 220080D2        mov     x2, #msg2_len
 0044 04000094        bl      print_string

                      // Выход из программы с помощью системного вызова 93 с возвращением состояния 0
 0048 000080D2        mov     x0, #0
 004c A80B80D2        mov     x8, #93
 0050 010000D4        svc     0

                  // Напечатаем строку; x1=адрес строки, x2=длина строки
                  print_string:
 0054 200080D2        mov     x0, #1
 0058 080880D2        mov     x8, #64
 005c 010000D4        svc     0
 0060 C0035FD6        ret     x30

                  // Преобразуем младшие 4 бита x0 в символ ascii в x0
                  nibble2ascii:
 0064 000C4092        and     x0, x0, #0xF
 0068 1F2800F1        cmp     x0, #10
 006c 64000054        bmi     lt10

 0070 00DC0091        add     x0, x0, #('A' - 10)
 0074 02000014        b       done

                  lt10:
 0078 00C00091        add     x0, x0, #'0'

                  done:
```

```
AARCH64 GAS   Ex__8_expr_arm64.s                                  page 2


  58 007c C0035FD6       ret     x30
  59
  60                     // Напечатаем байт в шестнадцатеричном формате
  61                     print_byte:
  62 0080 F5031EAA       mov     x21, x30
  63 0084 F40300AA       mov     x20, x0
  64 0088 00FC44D3       lsr     x0, x0, #4
  65 008c F6FFFF97       bl      nibble2ascii
  66 0090 C1010058       ldr     x1, =bytes
  67 0094 20140038       strb    w0, [x1], #1
  68
  69 0098 E00314AA       mov     x0, x20
  70 009c F2FFFF97       bl      nibble2ascii
  71 00a0 20000039       strb    w0, [x1]
  72
  73 00a4 21010058       ldr     x1, =bytes
  74 00a8 420080D2       mov     x2, #2
  75 00ac EAFFFF97       bl      print_string
  76
  77 00b0 FE0315AA       mov     x30, x21
  78 00b4 C0035FD6       ret     x30
  79
  80                     .data
  81                     msg1:
  82 0000 5B283132       .ascii  "[(129 - 66) * (445 + 136)] / 3 = "
  82      39202D20
  82      36362920
  82      2A202834
  82      3435202B
  83                     msg1_len = . - msg1
  84
  85                     bytes:
  86 0021 3F3F           .ascii  "??"
  87
  88                     msg2:
  89 0023 68             .ascii  "h"
  90                     msg2_len = . - msg2
```

Глава 11. Архитектура и набор инструкций RISC-V

Упражнение 1

Посетите сайт **https://www.sifive.com/software/** и скачайте пакет *Freedom Studio*. Это пакет разработки, основанный на интегрированной среде разработки (IDE) Eclipse, который оснащен полным набором инструментов для создания приложения RISC-V и его запуска на аппаратном процессоре RISC-V или в среде эмуляции, включенной в комплект Freedom Studio. Установите пакет, следуя указаниям в *руководстве пользователя Freedom Studio*. Запустите Freedom и создайте новый проект Freedom E SDK. В окне создания проекта в качестве цели выберите **qemu-sifive-u54** (это 64-разрядный одноядерный процессор RISC-V в конфигурации RV64GC). Выберите программу-пример `hello` и нажмите кнопку **Finish** (Готово). Запустится сборка программы-примера и эмулятора RISC-V. После завершения сборки отображается окно **Edit Configuration** (Изменение конфигурации). Нажмите **Debug** (Отладка), чтобы запустить программу в среде отладки эмулятора. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается строка "Hello, World!" ("Здравствуй, мир!").

Ответ. Установите пакет Freedom Studio, как описано выше. Обратите внимание, что путь к каталогу для вашей рабочей области не содержит пробелов. Запустите Freedom Studio:

1. В окне **Welcome to SiFive FreedomStudio! Let's Get Started...** (Добро пожаловать в SiFive FreedomStudio! Давайте начнем...) выберите **I want to create a new Freedom E SDK Project** (Я хочу создать новый проект Freedom E SDK).
2. В окне **Create a Freedom E SDK Project** (Создать новый проект Freedom E SDK) выберите в качестве цели **qemu-sifive-u54**.
3. Выберите программу-пример `hello`.
4. Нажмите кнопку **Finish** (Готово).
5. После завершения сборки отображается окно **Edit Configuration** (Изменение конфигурации).
6. Нажмите **Debug** (Отладка), чтобы запустить программу в среде отладки эмулятора.
7. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается строка "Hello, World!" ("Здравствуй, мир!").

Упражнение 2

Не закрывая проект из *упражнения 1*, в окне **Project** (Проект) найдите файл `hello.c` в папке `src`. Щелкните правой кнопкой мыши на файле и переименуйте его в `hello.s`. Откройте файл `hello.s` в редакторе и удалите все содержимое. Вставьте

программу на языке ассемблера, приведенную в *разд. "Язык ассемблера RISC-V" главы 11*. Выполните операцию очистки, затем снова соберите проект (чтобы запустить операцию очистки, нажмите комбинацию клавиш <Ctrl>+<9>). В меню **Run** (Выполнение) выберите команду **Debug** (Отладка). После запуска отладчика откройте окно для отображения исходного файла hello.s, окно **Disassembly** (Дизассемблирование) и окно **Registers** (Регистры). Раскройте дерево **Registers** (Регистры), чтобы отобразить регистры процессора RISC-V. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается строка "Привет, архитектор компьютеров!".

Ответ. Не закрывая проект из *упражнения 1*, найдите в окне **Project** (Проект) файл hello.c в папке src и сделайте следующее:

1. Щелкните правой кнопкой мыши на файле и переименуйте его в hello.s.
2. Откройте файл hello.s в редакторе и удалите все содержимое.
3. Вставьте программу на языке ассемблера, приведенную в *разд. "Язык ассемблера RISC-V" главы 11*. Это код на языке ассемблера, также доступный в файле Ex__2_riscv_assembly.s:

```
.section .text
.global main

main:
    # Резервирование места в стеке и сохранение адреса возврата

    addi    sp, sp, -16
    sd      ra, 0(sp)

    # Печать сообщения с помощью функции puts библиотеки C
1:  auipc   a0, %pcrel_hi(msg)
    addi    a0, a0, %pcrel_lo(1b)
    jal     ra, puts

    # Восстановление адреса возврата и sp, возврат к месту вызова
    ld      ra, 0(sp)
    addi    sp, sp, 16
    jalr    zero, ra, 0

.section .rodata
msg:
    .asciz "Привет, архитектор компьютеров!\n"
```

4. Выполните очистку, а затем снова соберите проект (чтобы запустить операцию очистки, нажмите комбинацию клавиш <Ctrl>+<9>).
5. В меню **Run** (Выполнение) выберите команду **Debug** (Отладка).
6. После запуска отладчика откройте окно для отображения исходного файла hello.s, окно **Disassembly** (Дизассемблирование) и окно **Registers** (Регистры).
7. Раскройте дерево **Registers** (Регистры), чтобы отобразить регистры процессора RISC-V.
8. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается строка "Привет, архитектор компьютеров!".

Упражнение 3

Напишите на ассемблере RISC-V программу, которая вычисляет следующее выражение и выводит результат в виде шестнадцатеричного числа: [(129 – – 66) × (445 + 136] : 3. В этой же программе создайте вызываемую функцию для печати одного байта в виде двух шестнадцатеричных цифр.

Ответ. Создайте новый проект Freedom Studio, используя действия, описанные в *упражнении 1 главы 11*. В окне **Project** (Проект) найдите файл hello.c в папке src:

1. Щелкните правой кнопкой мыши на файле и переименуйте его в hello.s.
2. Создайте свой исходный код на языке ассемблера в файле hello.s. Файл Ex__3_riscv_expr.s содержит следующий пример решения для этого упражнения:

```
.section .text
.global main

main:
    # Резервирование места в стеке и сохранение адреса возврата
    addi    sp, sp, -16
    sd      ra, 0(sp)

    # Напечатаем начальную строку вывода
    la      a0, msg1
    jal     ra, puts

    # Вычислим [(129 - 66) * (445 + 136)] / 3
    addi    a0, zero, 129
    addi    a0, a0, -66
    addi    a1, zero, 445
    add     a1, a1, 136
    mul     a0, a1, a0
```

```
    addi    a1, zero, 3
    divu    a0, a0, a1

    # Напечатаем старший байт результата
    sw      a0, 8(sp)
    srl     a0, a0, 8
    jal     ra, print_byte

    # Напечатаем младший байт результата
    lw      a0, 8(sp)
    jal     ra, print_byte

    # Напечатаем завершающую строку вывода
    la      a0, msg2
    jal     ra, puts

    # Восстановление адреса возврата и sp
    ld      ra, 0(sp)
    addi    sp, sp, 16

    # Обнуление кода завершения и возврат к месту вызова
    addi    a0, zero, 0
    ret

# Преобразование младших 4 бит в a0 в символ ascii в a0
nibble2ascii:
    # Резервирование места в стеке и сохранение адреса возврата
    addi    sp, sp, -16
    sd      ra, 0(sp)

    and     a0, a0, 0xF
    sltu    t0, a0, 10
    bne     t0, zero, lt10

    add     a0, a0, ('A' - 10)
    j       done

lt10:
    add     a0, a0, '0'
```

```
done:
    ld      ra, 0(sp)
    addi    sp, sp, 16
    ret

# Напечатаем байт в шестнадцатеричном формате
print_byte:
    # Резервирование места в стеке и сохранение адреса возврата
    addi    sp, sp, -16
    sd      ra, 0(sp)

    addi    t1, a0, 0
    srl     a0, t1, 4
    jal     ra, nibble2ascii
    la      t3, bytes
    sb      a0, 0(t3)

    addi    a0, t1, 0
    jal     nibble2ascii
    sb      a0, 1(t3)

    la      a0, bytes
    jal     ra, puts

    ld      ra, 0(sp)
    addi    sp, sp, 16
    ret

.section .data
msg1:
    .asciz  "[(129 - 66) * (445 + 136)] / 3 = "

bytes:
    .asciz  "??"

msg2:
    .asciz  "h"
```

3. Выполните очистку, а затем снова соберите проект (чтобы запустить операцию очистки, нажмите комбинацию клавиш <Ctrl>+<9>).
4. В меню **Run** (Выполнение) выберите команду **Debug** (Отладка).
5. После запуска отладчика откройте окно для отображения исходного файла `hello.s`, окно **Disassembly** (Дизассемблирование) и окно **Registers** (Регистры).
6. Раскройте дерево **Registers** (Регистры), чтобы отобразить регистры процессора RISC-V.
7. Проведите пошаговое выполнение программы и убедитесь, что в окне консоли отображается текст: `[(129 - 66) * (445 + 136)] / 3 = 2FA9h`.

Глава 12. Виртуализация процессоров

Упражнение 1

Скачайте и установите текущую версию VirtualBox. Скачайте, установите и запустите Ubuntu Linux в качестве виртуальной машины в VirtualBox.

Подключите гостевую ОС к Интернету с помощью мостового сетевого адаптера. Настройте и включите общий доступ к буферу обмена и файлам между гостевой системой Ubuntu и операционной системой хоста.

Ответ. Выполните следующие действия:

1. Скачайте установщик VirtualBox 6.1 (или более поздней версии) по адресу **https://www.virtualbox.org/wiki/Downloads**. Выберите версию, подходящую для операционной системы на хосте.
2. Запустите установщик VirtualBox и подтвердите предложенные по умолчанию настройки.
3. Скачайте образ 64-разрядной ОС Ubuntu Linux для VirtualBox. Вот один из источников такого образа: **https://www.osboxes.org/ubuntu/**. Если образ представлен в сжатом формате, распакуйте его. Если файл имеет расширение `7z`, используйте программу 7-Zip (**https://www.7-zip.org/**). После распаковки файл дискового образа для VirtualBox будет иметь расширение `vdi`.
4. Запустите программу VirtualBox Manager и щелкните на значке **New** (Создать). Дайте новой машине имя, например `Ubuntu`, выберите в качестве типа **Linux**, в качестве версии выберите **Ubuntu (64-bit)**. Щелкните **Next** (Далее).
5. В окне **Memory size** (Объем памяти) подтвердите предложенный по умолчанию объем памяти (или увеличьте его, если хотите).
6. В окне **Hard disk** (Жесткий диск) выберите **Use an existing virtual hard disk file** (Использовать существующий файл виртуального жесткого диска). Щелкните на кнопке **Browse** (Обзор) (она выглядит как папка), затем щелкните на кнопке **Add** (Добавить) в окне **Hard disk selector** (Выбор жесткого диска). Перейдите к скачанному файлу с расширением `vdi` и выберите **Open** (Открыть). Щелкните **Create** (Создать), чтобы завершить создание виртуальной машины.

7. Щелкните на значке **Settings** (Настройки) в VirtualBox. В разделе **General** (Общие) на вкладке **Advanced** (Дополнительно) выберите вариант **Bidirectional** (Двунаправленный) для параметра **Shared Clipboard** (Общий буфер обмена).
8. Щелкните **Network** (Сеть). На вкладке **Adapter 1** (Адаптер 1) рядом с полем **Attached to:** (Подключение) выберите **Bridged Adapter** (Мостовой адаптер).
9. В своей папке `Documents` (Документы) на диске Windows создайте папку с именем `share`. Щелкните **Shared Folders** (Общие папки) в окне **Settings** (Настройки) программы VirtualBox Manager для своей виртуальной машины Ubuntu. Щелкните на значке добавления общей папки (он выглядит как папка со знаком "плюс"). Выберите только что созданную на хосте папку `share` и щелкните **OK**.
10. Щелкните **OK** в окне **Settings** (Настройки), чтобы закрыть его.
11. Щелкните на значке **Start** (Пуск), чтобы запустить виртуальную машину. Когда загрузка системы Ubuntu завершится, войдите в систему с паролем `osboxes.org`.
12. После завершения процедуры входа в систему откройте окно терминала, нажав комбинацию клавиш <Ctrl>+<Alt>+<T>.
13. В окне терминала виртуальной машины установите программные пакеты с помощью следующих команд. Отвечайте на запросы, чтобы завершить установку:

```
sudo apt-get update
sudo apt-get install gcc make perl
sudo apt-get install build-essential linux-headers-`uname -r` dkms
```

14. В меню **Devices** (Устройства) окна виртуальной мащины Ubuntu выберите **Insert Guest Additions CD Image...** (Добавить образ Guest Additions...). Отвечайте на запросы, чтобы завершить установку. После завершения установки перезагрузите виртуальную машину.
15. Войдите в систему на виртуальной машине и откройте окно терминала. В терминале виртуальной машины создайте каталог с именем `share`, используя следующую команду:

```
mkdir share
```

16. Для того чтобы смонтировать общую папку, введите следующую команду в терминале виртуальной машины:

```
sudo mount -t vboxsf -o rw,uid=1000,gid=1000 share_folder ~/share
```

17. Создайте файл в общей папке системы Ubuntu:

```
cd ~/share
touch file1.txt
```

18. Убедитесь, что на хосте Windows в папке Documents\share находится файл с именем file1.txt.

Упражнение 2

В операционной системе Ubuntu, которую вы установили в *упражнении 1*, установите VirtualBox, затем установите и запустите версию виртуальной машины FreeDOS. Убедитесь в правильности выполнения команд DOS, таких как echo Hello World! и mem, в виртуальной машине FreeDOS. Выполнение этого упражнения означает, что вы реализовали экземпляр вложенной виртуализации.

Ответ

1. При неработающей виртуальной машине Ubuntu щелкните на значке **Settings** (Настройки) в VirtualBox Manager данной виртуальной машины. В разделе **System** (Система) на вкладке **Processor** (Процессор) установите флажок **Enable Nested VT-x/AMD-V** (Включить вложенную виртуализацию VT-x/AMD-V). Для полной поддержки этой функции требуется VirtualBox 6.1 или более поздней версии. Нажмите **OK** для сохранения изменений.
2. Запустите виртуальную машину Ubuntu. Войдите в виртуальную машину Ubuntu, откройте окно терминала и установите на нее VirtualBox с помощью следующей команды:

```
sudo apt-get install virtualbox
```

3. Установите на виртуальную машину Ubuntu программу 7-Zip с помощью следующей команды:

```
sudo apt-get install p7zip-full
```

4. Скачайте образ виртуального диска VirtualBox для FreeDOS по адресу **https://www.osboxes.org/freedos/**. Выполните следующие действия (при условии, что загруженный файл находится в каталоге ~/snap/firefox/common/Downloads и файл образа FreeDOS имеет имя 64-bit.7z):

```
cd
mkdir 'VirtualBox VMs'
cd 'VirtualBox VMs'
mv ~/snap/firefox/common/Downloads/64bit.7z .
7z x 64bit.7z
```

5. Запустите **VirtualBox** с помощью следующей команды:

```
virtualbox &
```

6. Создайте новую виртуальную машину в экземпляре **VirtualBox**, запущенном на виртуальной машине Ubuntu. Выберите следующие параметры:

```
Name: FreeDOS
Type: Other
Version: DOS
32MB RAM
Use an existing virtual hard disk file
```

7. Выберите файл VDI в каталоге `~/VirtualBox VMs` и завершите настройку виртуальной машины.
8. Щелкните на значке **Start** (Пуск) в VirtualBox Manager, чтобы запустить виртуальную машину FreeDOS.
9. После завершения загрузки виртуальной машины выполните в командной строке FreeDOS следующие команды:

```
echo Hello World!
mem
dir
```

На копии экрана на рис. П4 показан вывод результатов выполнения команды `mem`.

```
FreeDOS [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
C:\>mem

Memory Type         Total       Used       Free
----------------  ----------  --------  ----------
Conventional          639K        16K        623K
Upper                  72K        33K         39K
Reserved              313K       313K          0K
Extended (XMS)     31,680K       532K     31,148K
----------------  ----------  --------  ----------
Total memory       32,704K       894K     31,810K

Total under 1 MB      711K        49K        662K

Total Expanded (EMS)                  31M (32,571,392 bytes)
Free Expanded (EMS)                   30M (31,916,032 bytes)

Largest executable program size      623K (638,112 bytes)
Largest free upper memory block       26K ( 26,448 bytes)
FreeDOS is resident in the high memory area.
C:\>_
                                                    Right Ctrl
```

Рис. П4. *Копия экрана FreeDOS*

10. После завершения работы с FreeDOS закройте виртуальную машину с помощью следующей команды в командной строке FreeDOS:

```
shutdown
```

Упражнение 3

Создайте две отдельные копии гостевой машины Ubuntu в среде хоста VirtualBox. Настройте обе гостевые системы Ubuntu для подключения к *внутренней* сети VirtualBox. Задайте в этих двух машинах совместимые IP-адреса. Убедитесь, что каждая машина может получить ответ от другой, используя команду ping. Выполнение этого упражнения означает, что вы настроили в своей виртуализированной среде виртуальную сеть.

Ответ

1. В системе хоста VirtualBox откройте окно **Settings** (Настройки) для виртуальной машины Ubuntu, настроенной в *упражнении 1*, и выберите настройки **Network** (Сеть). Задайте для параметра типа сети **Attached to:** (Подключение) значение **Internal** (Внутренняя), затем щелкните **OK**.
2. Щелкните правой кнопкой мыши на виртуальной машине Ubuntu в VirtualBox Manager и выберите **Clone...** (Клонировать...) из контекстного меню. Щелкните **Next** (Далее) в меню **Clone VM** (Клонировать виртуальную машину). Оставьте **Full clone** (Полное клонирование) установленным и щелкните **Clone** (Клонировать). Дождитесь завершения процесса клонирования.
3. Откройте командную строку в хост-системе и перейдите в каталог установки VirtualBox. В Windows для перехода к месту установки по умолчанию можно использовать следующую команду:

```
cd "\Program Files\Oracle\VirtualBox"
```

4. Запустите DHCP-сервер для сети intnet VirtualBox с помощью следующей команды:

```
VBoxManage dhcpserver add --netname intnet --ip 192.168.10.1
--netmask 255.255.255.0 --lowerip 192.168.10.100
--upperip 192.168.10.199 --enable
```

5. Запустите обе виртуальные машины. На основе рекомендованных на предыдущем шаге настроек DHCP-сервера виртуальным машинам могут быть назначены IP-адреса 192.168.10.100 и 192.168.10.101.
6. Войдите в систему на обеих запущенных виртуальных машинах и откройте окно терминала в каждой из них.

Введите в каждом окне терминала следующую команду, чтобы отобразить системный IP-адрес:

```
hostname -I
```

7. Проверьте подключение к другой машине, используя команду ping. Например, если IP-адрес текущего машины равен 192.168.10.100, введите следующую команду:

```
ping 192.168.10.101
```

Выводимая на экран информация должна выглядеть следующим образом. Для того чтобы остановить обновление, нажмите комбинацию клавиш <Ctrl>+<C>.

```
osboxes@osboxes:~$ ping 192.168.10.101
PING 192.168.10.101 (192.168.10.101) 56(84) bytes of data.
64 bytes from 192.168.10.101: icmp_seq=1 ttl=64 time=0.372 ms
64 bytes from 192.168.10.101: icmp_seq=2 ttl=64 time=0.268 ms
64 bytes from 192.168.10.101: icmp_seq=3 ttl=64 time=0.437 ms
64 bytes from 192.168.10.101: icmp_seq=4 ttl=64 time=0.299 ms
^C
--- 192.168.10.101 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3054ms
rtt min/avg/max/mdev = 0.268/0.344/0.437/0.065 ms
osboxes@osboxes:~$
```

8. Выполните команду ping на второй машине, указав IP-адрес первой машины. Убедитесь, что реакция системы аналогична предыдущему результату.

Глава 13. Специализированные компьютерные архитектуры

Упражнение 1

Нарисуйте блок-схему вычислительной архитектуры для системы круглосуточных измерений и передачи данных о погоде с интервалом в 5 минут с помощью текстовых сообщений SMS. Система работает от аккумулятора и использует солнечные батареи для подзарядки аккумулятора в светлое время суток. Предположим, что метеорологические приборы потребляют минимальную среднюю мощность, требуя лишь кратковременного выхода на полную мощность в каждом цикле измерений.

Ответ. Исходя из требований, процессор, способный переходить в режим низкого энергопотребления в течение нескольких минут, должен работать от аккумулятора среднего размера в течение нескольких дней подряд. Энергопотребление можно свести к минимуму за счет включения метеорологических приборов только на время проведения измерений, а также включения приемопередатчика сотовой связи только для передачи данных.

На рис. П5 представлена одна из возможных конфигураций такой системы.

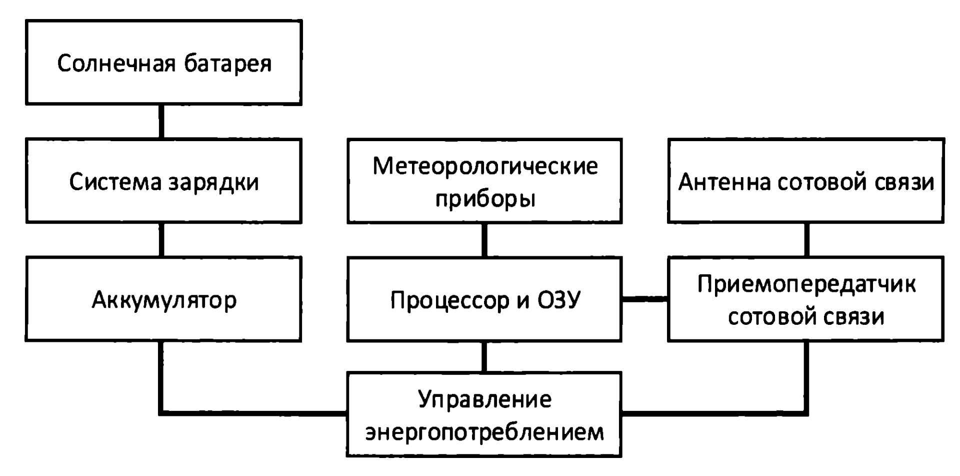


***Рис. П5.** Начальная схема системы сбора данных о погоде*

Упражнение 2

Для системы из *упражнения 1* определите подходящий доступный на рынке процессор и укажите причины, по которым этот процессор является хорошим выбором для решения данной задачи. Факторы, которые следует учитывать, включают стоимость, скорость обработки, устойчивость к суровым условиям, энергопотребление и встроенные компоненты, такие как оперативная память и интерфейсы для обмена данными.

Ответ. Выполните следующие действия:

1. Поиск в Интернете по фразе микропроцессор с малым энергопотреблением дает широкий выбор процессоров от таких производителей, как STM, Analog Devices, Texas Instruments, Microchip Technology и ряда других.
2. Поиск по фразе встроенный сотовый модем выдает список сотовых модемов, подходящих для решения данной задачи. Некоторые из этих устройств выполнены в виде **модульной системы** (system-on-module, SoM), объединяющей радиочастотный модем с программируемым процессорным ядром.
3. Модульная система MultiTech Dragonfly Nano (**https://www.multitech.com/brands/multiconnect-dragonfly-nano**), по-видимому, подходит для решения данной задачи. Это устройство продается за 103,95 доллара США и содержит про-

цессор ARM Cortex-M4 для пользовательских приложений. Модуль Dragonfly Nano предоставляет следующие интерфейсы ввода-вывода: последовательный UART, USB, I2C, SPI, 9 аналоговых входов и до 29 контактов для дискретных входов/выходов. Процессор Cortex-M4 содержит 1 Мбайт флеш-памяти и 128 Кбайт оперативной памяти.

4. В документации Dragonfly Nano сказано, что при ежедневной передаче небольшого объема данных устройство может годами работать от двух батареек типа AA.

5. Причины выбора Dragonfly Nano для решения этой задачи таковы.

   - **Цена.** Несмотря на то что цена более 100 долларов США для микропроцессорной платы считается высокой, непосредственная интеграция сотового модема решает ключевую задачу проектирования системы.
   - **Низкое энергопотребление.** В зависимости от потребностей метеорологических приборов в электропитании небольшая солнечная батарея в сочетании с перезаряжаемым аккумулятором небольшой емкости должна легко удовлетворить требования системы к питанию.
   - **Условия окружающей среды.** Диапазон рабочих температур для данной модульной системы составляет от –40 до +85 °С, что позволяет эксплуатировать систему в любой точке мира. Допустимый диапазон относительной влажности (от 20 до 90% относительной влажности, без конденсации) требует установки в защищенном от воздействия атмосферных факторов корпусе.

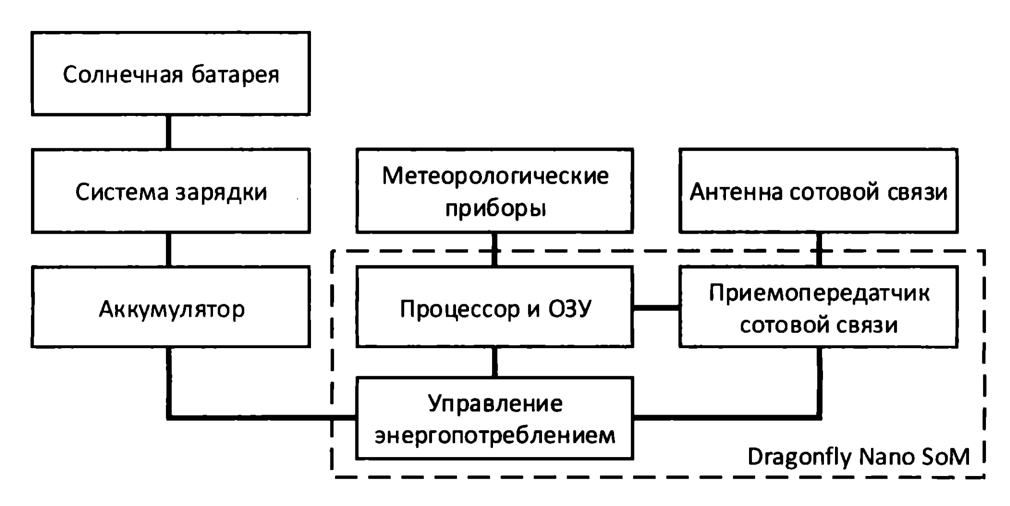


Рис. П6. Окончательная схема системы сбора данных о погоде

   - **Вычислительная мощность.** Данная модульная система содержит 32-разрядный процессор ARM STM32L471QG с тактовой частотой 80 МГц. Он обладает широкими возможностями, включая блок вычислений с плавающей запятой и динамическое масштабирование напряжения. Перед передачей данных можно выполнить обширную предварительную обработку измерений (фильтрацию, обнаружение неисправностей датчиков и т. д.). Объем флеш-

памяти и оперативной памяти в устройстве вполне достаточен для данной задачи.

- **Сертифицированное решение.** Модуль Dragonfly Nano сертифицирован FCC и операторами беспроводной связи для использования в сетях сотовой связи.
- **Поддержка разработки.** Бесплатные инструменты разработки и онлайн-ресурсы доступны по адресу **https://os.mbed.com/platforms/MTS-Dragonfly-Nano/**.

6. Пунктирная рамка на рис. П6 выделяет часть системы, реализованную с помощью модуля Dragonfly Nano.

Глава 14. Архитектуры для обеспечения кибербезопасности и конфиденциальности вычислений

Упражнение 1

Для всех своих учетных записей в Интернете, содержащих важные данные, настройте двухфакторную аутентификацию (где она поддерживается). К таким учетным записям относятся банковские счета, электронная почта, социальные сети, хранилища кода (если вы разработчик программного обеспечения), медицинские услуги и все остальное, что для вас ценно. На всех этапах этого процесса необходимо заручиться гарантией того, что вы используете информацию и программные приложения исключительно из надежных источников.

Ответ. Обширный список веб-сайтов с указанием поддержки (или отсутствия поддержки) двухфакторной аутентификации доступен на веб-сайте **2FA Directory** (**https://2fa.directory/**). 2FA — это сокращение фразы "двухфакторная аутентификация".

Наиболее распространенным методом реализации двухфакторной аутентификации является отправка сайтом SMS-сообщения с проверочным кодом на номер телефона, связанный с учетной записью после того, как пользователь введет действительное имя пользователя и пароль.

Код часто представляет собой последовательность из 6 цифр, которую пользователь должен ввести на веб-сайте для завершения процесса входа в систему. Два фактора, используемые для аутентификации, — это знание пользователем пароля учетной записи и продемонстрированный доступ к телефону, связанному с учетной записью.

Некоторые веб-сайты поддерживают приложение **Duo Mobile** (**https://duo.com/product/multi-factor-authentication-mfa/duo-mobile-app**) для двухфакторной аутентификации. При доступе к веб-сайту, использующему это приложение, после ввода

имени пользователя и пароля на вашем телефоне отображается уведомление. Для подтверждения доступа и завершения входа в систему достаточно одного касания экрана.

Упражнение 2

Для тех своих учетных записей в Интернете, которые содержат ценную информацию, но могут быть защищены двухфакторной аутентификацией, создайте надежные пароли. Надежный пароль должен быть длинным (15 символов или более) и содержать прописные и строчные буквы, цифры и специальные символы (например, `! " # $ % & ' ( ) * +`). Для того чтобы отслеживать эти сложные пароли, установите и используйте надежное приложение для хранения паролей. Будьте осторожны при выборе такого приложения и тщательно выбирайте его источник.

Ответ. Существует множество решений для безопасного хранения паролей на вашем компьютере и на других устройствах. Большинство браузеров, как и большинство пакетов антивирусных программ, предлагают решения для управления паролями. Также доступны отдельные приложения для управления паролями. Вы можете сузить свой выбор, выполнив поиск в Интернете по фразе `менеджер паролей`.

Когда сайт запрашивает новый пароль, вы можете использовать менеджер паролей, чтобы сгенерировать длинную строку произвольно выглядящих символов в качестве вашего нового пароля. Вам не потребуется запоминать пароль, т. к. он будет надежно сохранен менеджером паролей.

При выборе решения для управления паролями следует учитывать необходимость сохранения текущих паролей на всех ваших устройствах. Когда вы меняете пароль для веб-сайта, вам не захочется делать это в нескольких местах. Менеджер паролей на основе браузера, такого как **Firefox** (**https://www.mozilla.org/en-US/**), позаботится об этом за вас, если у вас есть учетная запись Firefox и вы выполнили вход в нее на каждом своем устройстве.

Упражнение 3

Обновите операционную систему, а также другие приложения и сервисы (например, Java) на всех компьютерах и других устройствах, находящихся под вашим контролем. Это послужит гарантией, что новые функции безопасности, включенные в эти обновления, начнут работать для вашей защиты вскоре после того, как станут доступны. Составьте план для продолжения регулярной установки обновлений по мере их выпуска, чтобы обеспечить свою защиту в будущем.

Ответ

1. Перейдите в настройки обновлений каждого своего устройства и проверьте, ожидают ли установки какие-либо обновления. Если такие обновления есть, установите их.

2. Если ожидающих установки обновлений нет, выберите в устройстве функцию проверки наличия обновлений и установите все доступные.
3. Запустите каждое приложение, которое вы используете и на которое полагаетесь, и также выберите функцию проверки наличия обновлений. Установите все доступные обновления.
4. Если в приложении есть функция автоматической проверки наличия обновлений, убедитесь, что она включена. Возможно, вы хотите получать уведомления о доступности обновлений, но не желаете, чтобы они устанавливались автоматически.
5. Настройте в своем приложении Календарь повторяющееся напоминание о необходимости проверять наличие обновлений для всех устройств и приложений с наименьшим интервалом, который вы считаете разумным, будь то еженедельно, раз в две недели или ежемесячно. Не выбирайте слишком длинные интервалы, т. к. ваши системы уязвимы в период между выявлением уязвимости и установкой обновления, которое ее исправляет.

Глава 15. Архитектуры блокчейна и майнинга биткоинов

Упражнение 1

Откройте веб-страницу для просмотра информации о транзакциях блокчейна по адресу **https://bitaps.com** и найдите список последних блоков. Щелкните на номере блока, откроется окно с заголовком блока в шестнадцатеричном формате и хешем SHA-256. Скопируйте оба эти элемента и напишите программу, чтобы определить, является ли предоставленный хеш правильным хешем заголовка. Не забудьте дважды выполнить алгоритм SHA-256 для вычисления хеша заголовка.

Ответ. Файл Python `Ex__1_compute_block_hash.py` содержит код хеширования заголовка блока:

```
#!/usr/bin/env python

"""Ex__1_compute_block_hash.py: ответ на упражнение 1 главы 15."""

# Это решение для биткоин-блока 711735.
# См. https://bitaps.com/711735

import binascii
import hashlib
```

```
# Заголовок блока, скопированный с сайта bitaps.com
header = '00000020505424e0dc22a7fb1598d3a048a31957315f' + \
    '737ec0d00b000000000000000000005f7fbc00ac45edd1f6ca7' + \
    '713f2b048d8a771c95e1afd9140d3a147a063f64a76781ea4' + \
    '61139a0c17f666fc1afdbc08'

# Хеш заголовка, скопированный с сайта bitaps.com
header_hash = \
    '0000000000000000000bc01913c2e05a5d38d39a9df0c8ba' + \
    '4269abe9777f41f'

# Отсечем все лишние байты, выходящие за пределы 80-байтового заголовка
header = header[:160]

# Преобразуем заголовок в двоичную форму
header = binascii.unhexlify(header)

# Вычислим хеш заголовка (выполним алгоритм SHA-256 дважды)
computed_hash = hashlib.sha256(header).digest()
computed_hash = hashlib.sha256(computed_hash).digest()

# Поменяем порядок байтов на обратный
computed_hash = computed_hash[::-1]

# Преобразуем двоичный хеш заголовка в строку шестнадцатеричных символов
computed_hash = \
    binascii.hexlify(computed_hash).decode("utf-8")

# Выведем результат
print('Хеш заголовка:    ' + header_hash)
print('Вычисленный хеш: ' + computed_hash)

if header_hash == computed_hash:
    result = 'Хеши совпадают!'
else:
    result = ' Хеши НЕ СОВПАДАЮТ!'

print(result)
```

Для того чтобы выполнить эту программу, при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__1_compute_block_hash.py
```

Это результат выполнения программы:

```
C:\>python Ex__1_compute_block_hash.py
Хеш заголовка:
00000000000000000000bc01913c2e05a5d38d39a9df0c8ba4269abe9777f41f
Вычисленный хеш:
00000000000000000000bc01913c2e05a5d38d39a9df0c8ba4269abe9777f41f
Хеши совпадают!
```

Упражнение 2

Настройте полный равноправный узел Bitcoin и подключите его к сети Bitcoin. Загрузите программное обеспечение Bitcoin Core по адресу **https://bitcoin.org/en/download**. Вам потребуется быстрое подключение к Интернету и не менее 200 Гбайт свободного места на диске.

Ответ

1. Скачайте установщик ПО Bitcoin Core, доступный по адресу **https://bitcoin.org/en/download**.
2. После завершения установки запустите приложение Bitcoin Core. Приложение запустит скачивание всего блокчейна Bitcoin, начиная с первичного блока, добавленного в 2009 г., и заканчивая самым последним добавленным блоком. Этот процесс может занять несколько часов или дней в зависимости от пропускной способности вашего подключения к Интернету.
3. Для проведения первоначальной проверки приложению Bitcoin Core потребуется около 200 Гбайт дискового пространства, однако после этого объем хранимых данных можно будет уменьшить до выбранного вами предела, который по умолчанию составляет 2 Гбайт.
4. После скачивания блокчейна узел перейдет к работе в качестве полного равноправного узла сети. Вы сможете отображать подключения приложения к равноправным узлам сети и отслеживать добавление новых транзакций в пул транзакций, ожидающих включения в следующий блок, который будет добавлен в блокчейн.
5. В этом приложении вы также можете создать биткоин-кошелек и использовать его для проведения собственных биткоин-транзакций. Если вы используете это

приложение для хранения значительного количества биткоинов, необходимо применять наилучшие доступные методы всесторонней защиты операционной системы главного компьютера и его приложений, чтобы гарантировать, что они не будут взломаны с целью кражи ваших денег.

Глава 16. Архитектуры для самоуправляемых автомобилей

Упражнение 1

Если на вашем компьютере еще не установлен Python, перейдите по адресу **https://www.python.org/downloads/** и установите текущую версию. Убедитесь, что файлы Python включены в пути поиска, набрав `python --version` в командной строке системы. Вы должны получить ответ, подобный следующему: `Python 3.10.3`. Установите TensorFlow (платформу для машинного обучения с открытым исходным кодом) с помощью команды (также в системной командной строке) `pip install tensorflow`. Возможно, для успешной установки вам потребуется использовать функцию запуска от имени администратора при открытии окна командной строки. Установите Matplotlib (библиотеку для визуализации данных) с помощью команды `pip install matplotlib`.

Ответ. Пакетный файл Windows `Ex__1_install_tensorflow.bat` содержит команды для установки TensorFlow и Matplotlib:

```
REM Ex__1_install_tensorflow.bat: Answer to Ch 16 Ex 1.

REM This batch file installs TensorFlow and Matplotlib in Windows.
REM Python must be installed (see https://www.python.org/downloads/).
REM The Python installation directory must be in the system path.

python --version

pip install tensorflow

pip install matplotlib
```

Для того чтобы выполнить пакетный файл при условии, что Python установлен и находится по известному системе пути, откройте окно командной строки от имени *администратора* и выполните команду `Ex__1_install_tensorflow.bat`.

Упражнение 2

Используя библиотеку TensorFlow, создайте программу, которая загружает набор данных CIFAR-10 и отображает подмножество изображений вместе с метками, связанными с каждым изображением. Этот набор данных является продуктом **Канадского института перспективных исследований** (Canadian Institute for Advanced Research, CIFAR) и содержит 60 000 изображений, каждое из которых состоит из 32×32 RGB-пикселов. Эти изображения были случайным образом разделены на обучающий набор из 50 000 изображений и тестовый набор из 10 000 изображений. Каждое изображение было помечено людьми как представляющее предмет одной из 10 категорий: самолет, автомобиль, птица, кошка, олень, собака, лягушка, лошадь, судно или грузовик. Для получения дополнительной информации о наборе данных CIFAR-10 см. технический отчет Алекса Крижевски (Alex Krizhevsky) по адресу **https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf**.

Ответ. Файл Python `Ex__2_load_dataset.py` содержит код для загрузки набора данных и отображения подмножества изображений:

```
#!/usr/bin/env python

"""Ex__2_load_dataset.py: ответ на упражнение 2 главы 16."""

from tensorflow.keras import datasets
import matplotlib.pyplot as plt

def load_dataset():
    (train_images, train_labels), \
        (test_images, test_labels) = \
        datasets.cifar10.load_data()

    # Приведем значения пикселов к диапазону 0-1
    train_images = train_images / 255.0
    test_images = test_images / 255.0

    return train_images, train_labels, \
        test_images, test_labels

def plot_samples(train_images, train_labels):
    class_names = ['Airplane', 'Automobile', 'Bird',
                   'Cat', 'Deer','Dog', 'Frog',
                   'Horse', 'Ship', 'Truck']
```

```
    plt.figure(figsize=(14,7))
    for i in range(60):
        plt.subplot(5,12,i + 1)
        plt.xticks([])
        plt.yticks([])
        plt.imshow(train_images[i])
        plt.xlabel(class_names[train_labels[i][0]])

    plt.show()

if __name__ == '__main__':
    train_images, train_labels, \
        test_images, test_labels = load_dataset()
    plot_samples(train_images, train_labels)
```

Для того чтобы выполнить эту программу при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__2_load_dataset.py
```

Если вы получите сообщение об ошибке с текстом `cudart64_110.dll not found`, можете спокойно проигнорировать его. Оно означает, что у вас не установлена библиотека для запуска TensorFlow на графическом процессоре Nvidia CUDA. Вместо него данный код будет выполняться (более медленно) на вашем системном процессоре.

На рис. П7 приведен набор примеров изображений, выводимых кодом.



Рис. П7. *Примеры изображений набора данных CIFAR*

Упражнение 3

Используя библиотеку TensorFlow, создайте программу, которая строит сверточную нейросеть (CNN) на основе структуры, показанной на рис. 16.1. Примените фильтр свертки 3 × 3 в каждом сверточном слое. Используйте 32 фильтра в первом сверточном слое и 64 фильтра в двух других сверточных слоях. В скрытом слое используйте 64 нейрона. Выделите 10 выходных нейронов, отражающих отнесение изображения к одной из 10 категорий CIFAR-10.

Ответ. На рис. П8 показана структура CNN, аналогичная рис. 16.1.



Рис. П8. *Структура CNN для классификации изображений*

Файл Python `Ex__3_create_network.py` содержит код для создания этой модели CNN:

```
#!/usr/bin/env python

"""Ex__3_create_network.py: ответ на упражнение 3 главы 16."""

from tensorflow.keras import datasets, layers, models, \
    optimizers, losses
```

```
def load_dataset():
    (train_images, train_labels), \
        (test_images, test_labels) = \
        datasets.cifar10.load_data()

    # Приведем значения пикселов к диапазону 0-1
    train_images = train_images / 255.0
    test_images = test_images / 255.0

    return train_images, train_labels, \
        test_images, test_labels

def create_model():
    # Каждое изображение имеет размер 32x32 пиксела и 3 цветовые плоскости RGB
    image_shape = (32, 32, 3)

    # Размер ядра сверточного фильтра составляет 3x3 пиксела
    conv_filter_size = (3, 3)

    # Количество сверточных фильтров в каждом слое
    filters_layer1 = 32
    filters_layer2 = 64
    filters_layer3 = 64

    # Выполним группировку с выбором максимума в области 2x2 пиксела
    pooling_size = (2, 2)

    # Количество нейронов в каждом плотном слое
    hidden_neurons = 64
    output_neurons = 10

    model = models.Sequential([
        # Первый сверточный слой, за которым следует группировка с выбором максимума
        layers.Conv2D(filters_layer1, conv_filter_size, \
            activation='relu', input_shape=image_shape),
        layers.MaxPooling2D(pooling_size),
```

```
        # Второй сверточный слой, за которым следует группировка с выбором максимума
        layers.Conv2D(filters_layer2, conv_filter_size, \
            activation='relu'),
        layers.MaxPooling2D(pooling_size),

        # Третий сверточный слой, за которым следует сглаживание
        layers.Conv2D(filters_layer3, conv_filter_size, \
            activation='relu'),
        layers.Flatten(),

        # Плотный слой, за которым следует выходной слой
        layers.Dense(hidden_neurons, activation='relu'),
        layers.Dense(output_neurons)
    ])

    model.compile(optimizer=optimizers.Adam(), \
        loss=losses.SparseCategoricalCrossentropy( \
            from_logits=True), metrics=['accuracy'])

    return model

if __name__ == '__main__':
    train_images, train_labels, test_images, \
        test_labels = load_dataset()
    model = create_model()
    model.summary()
```

Для того чтобы выполнить эту программу при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__3_create_network.py
```



ПРИМЕЧАНИЕ

Вы можете игнорировать любые предупреждающие сообщения об отсутствии графического процессора, если в вашей системе его нет. Если графический процессор не настроен для использования с TensorFlow, этот код будет выполняться на системном процессоре.

Вот результат выполнения программы:

```
C:\>Ex__3_create_network.py
2021-12-12 19:26:07.938984: I tensorflow/core/platform/cpu_feature_guard.
cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural
Network Library (oneDNN) to use the following CPU instructions in
performance-critical operations:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the
appropriate compiler flags.
2021-12-12 19:26:08.282366: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1525] Created device
/job:localhost/replica:0/task:0/
device:GPU:0 with 3617 MB memory:  -> device: 0, name: Quadro P2200, pci bus id:
0000:01:00.0, compute capability: 6.1
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________

C:\>
```

Упражнение 4

Используя библиотеку TensorFlow, создайте программу, которая обучает нейросеть CNN, разработанную в *упражнении 3*, и испытайте итоговую модель с помощью тестовых изображений набора CIFAR-10. Определите долю правильно идентифицированных нейросетью тестовых изображений.

Ответ. Файл Python `Ex__4_train_model.py` содержит код для создания, обучения и тестирования модели CNN:

```
#!/usr/bin/env python

"""Ex__4_train_model.py: ответ на упражнение 4 главы 16."""

from tensorflow.keras import datasets, layers, models, optimizers, losses
import matplotlib.pyplot as plt

def load_dataset():
    (train_images, train_labels), \
        (test_images, test_labels) = \
        datasets.cifar10.load_data()

    # Приведем значения пикселов к диапазону 0-1
    train_images = train_images / 255.0
    test_images = test_images / 255.0

    return train_images, train_labels, \
           test_images, test_labels

def create_model():
    # Каждое изображение имеет размер 32x32 пиксела и 3 цветовые плоскости RGB
    image_shape = (32, 32, 3)

    # Размер ядра сверточного фильтра составляет 3x3 пиксела
    conv_filter_size = (3, 3)

    # Количество сверточных фильтров в каждом слое
    filters_layer1 = 32
    filters_layer2 = 64
    filters_layer3 = 64
```

```
    # Выполним группировку с выбором максимума в области 2x2 пиксела
    pooling_size = (2, 2)

    # Количество нейронов в каждом плотном слое
    hidden_neurons = 64
    output_neurons = 10

    model = models.Sequential([
        # Первый сверточный слой, за которым следует группировка с выбором максимума
        layers.Conv2D(filters_layer1, conv_filter_size, \
            activation='relu', input_shape=image_shape),
        layers.MaxPooling2D(pooling_size),

        # Второй сверточный слой, за которым следует группировка с выбором максимума
        layers.Conv2D(filters_layer2, conv_filter_size, \
            activation='relu'),
        layers.MaxPooling2D(pooling_size),

        # Третий сверточный слой, за которым следует сглаживание
        layers.Conv2D(filters_layer3, conv_filter_size, \
            activation='relu'),
        layers.Flatten(),

        # Плотный слой, за которым следует выходной слой
        layers.Dense(hidden_neurons, activation='relu'),
        layers.Dense(output_neurons)
    ])

    model.compile(optimizer=optimizers.Adam(),
        loss=losses.SparseCategoricalCrossentropy( \
            from_logits=True), metrics=['accuracy'])

    return model

def train_model(train_images, train_labels, \
            test_images, test_labels, model):
    history = model.fit(train_images, train_labels,
        epochs=10, validation_data=(test_images, test_labels))
```

```
    test_loss, test_acc = model.evaluate(test_images, \
        test_labels, verbose=2)

    return history, test_acc

def plot_model_accuracy(history):
    plt.figure()
    plt.plot(history.history['accuracy'], label='Точность')
    plt.plot(history.history['val_accuracy'],
        label = Точность проверки')
    plt.xlabel('Эпоха')
    plt.ylabel('Точность')
    plt.ylim([0.5, 1])
    plt.legend(loc='upper left')
    plt.grid()
    plt.show()

if __name__ == '__main__':
    train_images, train_labels, test_images, \
        test_labels = load_dataset()
    model = create_model()
    history, test_acc = train_model(train_images, \
        train_labels, test_images, test_labels, model)
    print()
    print('='*31)
    print('| Точность проверки:    {:.2f}% |'.
          format(100*test_acc))
    print('='*31)
    plot_model_accuracy(history)
```

Для того чтобы выполнить эту программу при условии, что Python установлен и находится по известному системе пути, выполните следующую команду:

```
python Ex__4_train_model.py
```



ПРИМЕЧАНИЕ

Вы можете игнорировать любые предупреждающие сообщения об отсутствии графического процессора, если в вашей системе его нет. Если графический процессор не настроен для использования с TensorFlow, этот код будет выполняться на системном процессоре.

Ваши результаты должны указывать на точность примерно 70%. Для такой простой CNN это огромное улучшение по сравнению с точностью случайного угадывания, которая составила бы 10%.

Вот результат запуска программы:

```
2021-12-12 17:55:19.402677: I tensorflow/core/platform/cpu_feature_guard.cc:151]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical operations:
AVX AVX2

To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.

2021-12-12 17:55:19.802026: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1525] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 3617 MB memory:  -> device: 0,
name: Quadro P2200, pci bus id: 0000:01:00.0, compute capability: 6.1

Epoch 1/10

2021-12-12 17:55:21.475358: I tensorflow/stream_executor/cuda/cuda_dnn.cc:366]
Loaded cuDNN version 8301

1563/1563 [==============================] - 9s 5ms/step - loss: 1.5032 -
accuracy: 0.4521 - val_loss: 1.2326 - val_accuracy: 0.5559

Epoch 2/10

1563/1563 [==============================] - 7s 5ms/step - loss: 1.1306 -
accuracy: 0.5996 - val_loss: 1.0361 - val_accuracy: 0.6318

Epoch 3/10

1563/1563 [==============================] - 8s 5ms/step - loss: 0.9704 -
accuracy: 0.6589 - val_loss: 1.0053 - val_accuracy: 0.6517

Epoch 4/10

1563/1563 [==============================] - 7s 5ms/step - loss: 0.8831 -
accuracy: 0.6904 - val_loss: 0.8999 - val_accuracy: 0.6883

Epoch 5/10

1563/1563 [==============================] - 7s 5ms/step - loss: 0.8036 -
accuracy: 0.7177 - val_loss: 0.8924 - val_accuracy: 0.6956

Epoch 6/10

1563/1563 [==============================] - 7s 5ms/step - loss: 0.7514 -
accuracy: 0.7374 - val_loss: 0.9180 - val_accuracy: 0.6903

Epoch 7/10

1563/1563 [==============================] - 7s 5ms/step - loss: 0.7020 -
accuracy: 0.7548 - val_loss: 0.8755 - val_accuracy: 0.7074

Epoch 8/10

1563/1563 [==============================] - 7s 5ms/step - loss: 0.6599 -
accuracy: 0.7694 - val_loss: 0.8505 - val_accuracy: 0.7116

Epoch 9/10

1563/1563 [==============================] - 8s 5ms/step - loss: 0.6180 -
accuracy: 0.7842 - val_loss: 0.8850 - val_accuracy: 0.7058
```

```
Epoch 10/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.5825 -
accuracy: 0.7943 - val_loss: 0.8740 - val_accuracy: 0.7128
313/313 - 1s - loss: 0.8740 - accuracy: 0.7128 - 648ms/epoch - 2ms/step

==============================
| Точность проверки:    71.28% |
==============================
```

На рис. П9 отражена точность классификации CNN на обучающих изображениях (**Точность**) и тестовых изображениях (**Точность проверки**) после каждой из 10 эпох обучения.

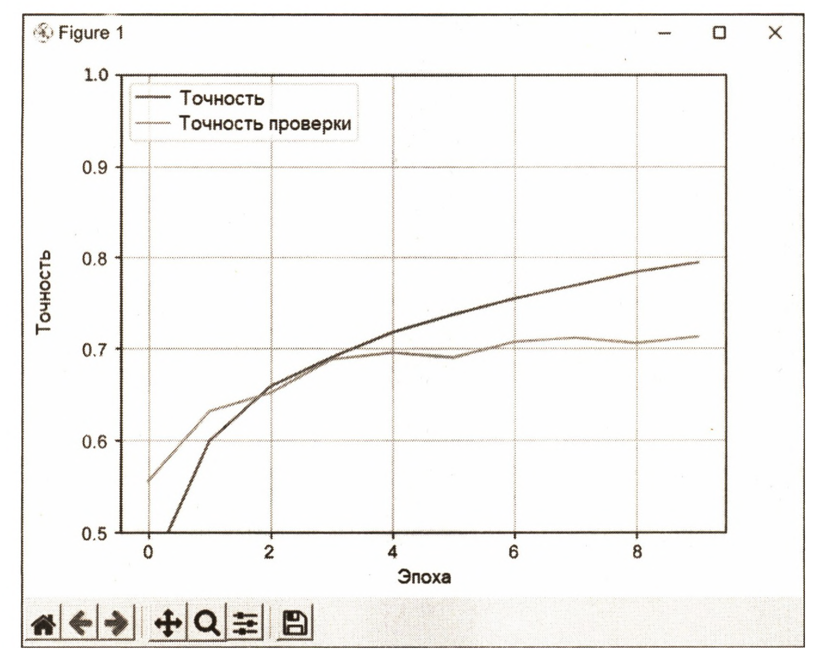


Рис. П9. *Точность классификации изображений сетью CNN*

Глава 17. Квантовые вычисления и другие перспективные направления в вычислительных архитектурах

Упражнение 1

Установите платформу разработки программного обеспечения для квантовых процессоров Qiskit, следуя указаниям по адресу: **https://qiskit.org/documentation/getting_started.html**. Эти указания предполагают установку набора инструментов

для обработки данных и машинного обучения Anaconda (**https://www.anaconda.com/**). После установки пакета Anaconda создайте виртуальную среду Conda с именем qisketenv, где будет проходить ваша работа над квантовым кодом, и установите Qisket в этой среде с помощью команды `pip install qiskit`. Убедитесь, что вы установили дополнительные зависимости визуализации с помощью команды `pip install qiskit-terra[visualization]`.

Ответ

1. Загрузите установщик Anaconda по адресу **https://www.anaconda.com/distribution/**. Выберите текущую версию и соответствующий 32- или 64-разрядный вариант для вашего главного компьютера.
2. Запустите установщик Anaconda и подтвердите предложенные по умолчанию настройки. После завершения установки закройте программу установки.
3. Запустите Anaconda из строки поиска Windows, набрав anaconda и щелкнув на **Anaconda Prompt**, когда эта фраза обнаружится в списке поиска. Отобразится окно консоли.
4. С помощью приглашения Anaconda создайте и активируйте виртуальную среду с именем qiskitenv посредством следующих команд (установите все рекомендуемые пакеты):

```
conda create -n qiskitenv python=3.8
conda activate qiskitenv
```

5. Установите Qiskit и зависимости визуализации с помощью следующих команд:

```
pip install qiskit
pip install qiskit-terra[visualization]
```

6. На этом установка завершена.

Упражнение 2

Создайте бесплатную учетную запись IBM Quantum по адресу: **https://quantum-computing.ibm.com/**. Найдите свой токен для IBM Quantum Services API на веб-сайте **https://quantum-computing.ibm.com/account** и установите его в свою локальную среду, используя указания по адресу **https://qiskit.org/documentation/stable/0.24/install.html**.

Ответ

1. Посетите веб-сайт **https://quantum-computing.ibm.com/**. Если у вас еще нет учетной записи, щелкните по ссылке **Create an IBMid account** (Создать учетную запись IBMid), чтобы ее создать.

2. После входа в систему найдите на экране поле **API token**. Нажмите кнопку, чтобы скопировать свой API-токен в буфер обмена.
3. Вернитесь к командной строке Anaconda для работы со средой qiskitenv, которую вы создали в *упражнении 1*.
4. Введите в командной строке Anaconda следующие команды, чтобы настроить свой API-токен (вам нужно будет заменить MY_TOKEN на тот, который вы скопировали в буфер обмена на *шаге 2*):

```
python
import qiskit
from qiskit import IBMQ
IBMQ.save_account('MY_TOKEN')
```

Упражнение 3

Рассмотрите пример квантовой программы на веб-сайте **https://qiskit.org/documentation/tutorials/circuits/1_getting_started_with_qiskit.html**. В этом примере создается квантовая схема, содержащая три кубита, которая реализует состояние **Гринбергера — Хорна — Цайлингера** (ГХЦ — Greenberger — Horne — Zeilinger, GHZ). Состояние ГХЦ проявляет ключевые свойства квантовой запутанности. Выполните код в среде моделирования на своем компьютере.

Ответ

1. Запустите консоль командной строки Anaconda. Введите anaconda в поле поиска Windows и щелкните на **Anaconda prompt**, когда эта фраза появится в списке поиска. Отобразится окно консоли.
2. Войдите в среду qiskitenv с помощью следующей команды:

```
conda activate qiskitenv
```

3. Введите в командной строке Anaconda следующие команды:

```
python
import numpy as np
from qiskit import *
```

4. Создайте квантовую схему с тремя кубитами в состоянии ГХЦ и добавьте измерения для каждого кубита:

```
circ = QuantumCircuit(3)
# Добавим затвор H для кубита 0 для создания суперпозиции
circ.h(0)
```

```
# Добавим затвор CX (CNOT). Кубит 0 - управляющий, кубит 1 - цель
circ.cx(0,1)
# Добавим затвор CX (CNOT). Кубит 0 - управляющий, кубит 2 - цель
circ.cx(0,2)

# Добавим измерение для каждого из кубитов
meas = QuantumCircuit(3, 3)
meas.barrier(range(3))
meas.measure(range(3),range(3))

# Объединим две схемы
circ.add_register(meas.cregs[0])
qc = circ.compose(meas)
```

5. **Выведите схему на экран:**

```
qc.draw()
```

Результаты выполнения этой команды должны выглядеть следующим образом:

```
>>> qc.draw()
        ┌───┐           ░ ┌─┐
   q_0: ┤ H ├──■────■───░─┤M├──────
        └───┘┌─┴─┐  │   ░ └╥┘┌─┐
   q_1: ─────┤ X ├──┼───░──╫─┤M├───
             └───┘┌─┴─┐ ░  ║ └╥┘┌─┐
   q_2: ──────────┤ X ├─░──╫──╫─┤M├
                  └───┘ ░  ║  ║ └╥┘
   c: 3/═══════════════════╩══╩══╩═
                           0  1  2
>>>
```

6. **Запустите схему на своем компьютере, используя имитатор `qasm_simulator`. Параметр `shots` обеспечивает подсчет количества проходов выполнения схемы для сбора статистических результатов:**

```
backend_sim = Aer.get_backend('qasm_simulator')
job_sim = backend_sim.run(transpile(qc, backend_sim), shots=1024)
```

7. Извлеките и отобразите количество появлений каждого битового шаблона в результате проведения моделирования:

```
result_sim = job_sim.result()
counts_sim = result_sim.get_counts(qc)
counts_sim
```

8. Должны быть выведены результаты, похожие (но не идентичные) на следующие:

```
>>> counts_sim
{'111': 506, '000': 518}
>>>
```

Упражнение 4

Выполните код из *упражнения 3* на квантовом компьютере IBM.

Ответ

1. Повторите *шаги 1–5* из *упражнения 3*, чтобы создать квантовую схему.
2. Импортируйте информацию о своей учетной записи IBMQ и выведите список доступных поставщиков ресурсов для квантовых вычислений:

```
from qiskit import IBMQ
IBMQ.load_account()
provider = IBMQ.get_provider(group='open')
provider.backends()
```

3. Если вы посетите веб-страницу IBM Quantum по адресу **https://quantum-computing.ibm.com/**, вы сможете увидеть длину очередей заданий для доступных квантовых компьютеров. Выберите систему с достаточным для вашей схемы количеством кубитов и короткой очередью заданий. В этом примере предполагается, что вы выбрали компьютер `ibmq_bogota`.
4. Добавьте свое задание в очередь и следите за его состоянием с помощью следующих команд. Параметр `shots` обеспечивает подсчет количества проходов выполнения схемы для сбора статистических результатов.

```
backend = provider.get_backend('ibmq_bogota')
from qiskit.tools.monitor import job_monitor
job_exp = execute(qc, backend=backend, shots=1024)
job_monitor(job_exp)
```

После завершения прогона вы увидите следующую строку вывода:

```
Job Status: job has successfully run
```

5. После завершения задания извлеките результаты с помощью следующей команды:

```
result_exp = job_exp.result()
```

6. Извлеките и отобразите количество появлений каждого битового шаблона в результате прогона квантового компьютера:

```
counts_exp = result_exp.get_counts(qc)
counts_exp
```

Примерно в 50% случаев выходная битовая строка для этой схемы должна быть равна 000, а в остальных 50% случаев она должна быть равна 111. Однако эти системы являются **"зашумленнными" квантовыми компьютерами среднего масштаба** (noisy, intermediate-scale quantum, NISQ). Должны быть выведены результаты, похожие (но не идентичные) на следующие:

```
>>> counts_exp
{'000': 467, '001': 15, '010': 23, '011': 17, '100': 21, '101': 127, '110': 16,
'111': 338}
>>>
```

Предметный указатель

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

А

Б

В

Г

Д

Е

З

И

К

Л

М

Н

Р

С

Т

У

Ф



www.bhv.ru



Ревич Ю.

Занимательная электроника. — 7-е изд., перераб. и доп.

Отдел оптовых поставок:
e-mail: opt@bhv.ru



- Начала начал электроники
- Оборудуем домашнюю лабораторию
- Транзисторы, резисторы, конденсаторы, дисплеи
- Операционные усилители, импульсные источники питания
- Аналоговые и логические схемы, микроконтроллеры
- Arduino — электроника для домашнего мастера
- Импортозамещение по-китайски: улучшенный аналог Arduino

На практических примерах рассказано о том, как проектировать, отлаживать и изготавливать электронные устройства в домашних условиях. От физических основ электроники, описания устройства и принципов работы различных радиоэлектронных компонентов, советов по оборудованию домашней лаборатории автор переходит к конкретным аналоговым и цифр вым схемам, включая устройства на основе микроконтроллеров. Приведены элементарные сведения по метрологии и теоретическим основам электроники. Дано множество практических рекомендаций: от принципов правильной организации электропитания до разводки плат и приобретения компонентов применительно к российским условиям. В 7-м издании обновлены многие разделы, содержавшие устаревшие сведения, подробнее рассказано об источниках тока для осветительных приборов, измерениях электрических величин, генераторных схемах, импульсных источниках питания, добавлены новые примеры применения платформы Arduino, а также приведен пример усовершенствованного контроллера китайского производства для замены Arduino.

Ревич Юрий Всеволодович, инженер-электронщик с многолетним стажем. Занимался автоматизацией производств, конструированием измерительных приборов для изучения океана и других научных исследований. Начиная с 2000-х — IT-обозреватель и журналист научно-популярной тематики. Основной круг интересов — информационные технологии, их влияние на современное общество, история компьютеров и электронных приборов. Имеет несколько сотен публикаций в журналах, газетах и сетевых изданиях. Автор более 10 книг, среди которых «Программирование микроконтроллеров AVR: от Arduino к ассемблеру», два сборника «Информационные технологии в СССР» и др.

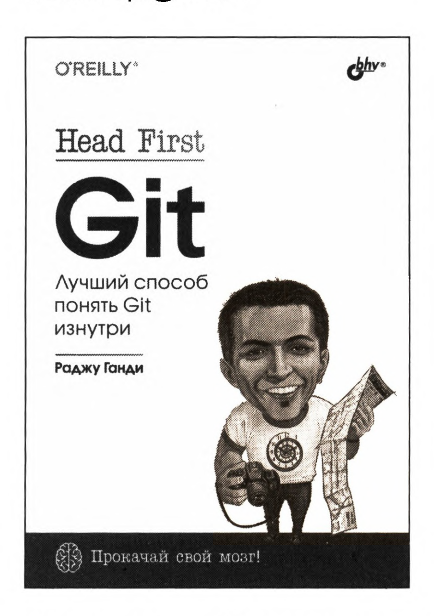


www.bhv.ru

Ганди Р.

Head First. Git

Отдел оптовых поставок:
e-mail: opt@bhv.ru



Книга поможет быстро и легко изучить самый популярный в мире инструмент контроля версий Git. В ней использована уникальная методика Head First, выходящая за рамки синтаксиса и инструкций по решению конкретных задач, а эффектное визуальное оформление разработано с учетом того, как работает и наиболее продуктивно усваивает информацию мозг.

Рассмотрены основы Git, свойства ветвления кода, слияние, коммиты, устройство репозитория Git и поиск в нем, отмена действий и исправление ошибок. Особое внимание уделено командной работе с Git, передовым методам взаимодействия и советам профессионалов по эффективной работе.

iOS и iPhone глазами хакера



- Средства безопасностии новые возможности в iOS 16
- Особенности распространения троянов для iOS
- Шпионаж за пользователем при помощи выключенного iPhone
- Эксплойт checkm8 и его использование
- Джейлбрейк — как сделать и зачем он нужен?
- Неофициальные твики и приложения для iOS с джейлбрейком
- Альтернативы для Cydia
- Установка сторонних приложений на iPhone без джейлбрейка
- Настройка VPN на iPhone
- Автоматизация iOS с помощью инструмента «Команды»
- Установка альтернативных ОС на iPhone с помощью виртуальных машин
- Написание программ на Python в среде разработки Pyto

Эта книга — сборник лучших, тщательно отобранных статей из легендарного журнала «Хакер». Рассмотрена система безопасности iOS 16, методы сбора и передачи информации устройствами Apple при выключенном питании. Рассказано о методах взлома файловой системы (джейлбрейк), об уязвимости эксплоита checkm8, об установке твиков и сторонних приложений на iPhone. Раскрыты методы установки стороннего ПО на мобильные устройства с iOS без джейлбрейка. Приведены альтернативы менеджеру пакетов Cydia, рассказано об использовании виртуальных машин в iOS. Представлены способы настройки VPN и автоматизации iOS при помощи инструмента «Команды». Приведен обзор возможностей Pyto — среды программирования на Python для мобильных устройств Apple.

1

Введение в архитектуру компьютеров

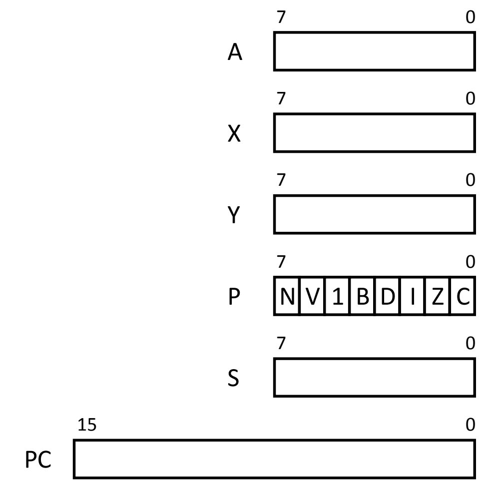


Рис. 1.1. *Набор регистров процессора 6502*

2

Цифровая логика

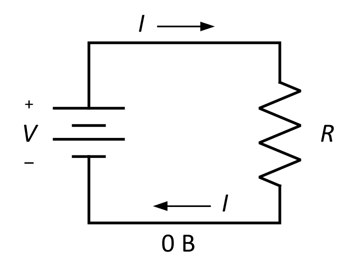


Рис. 2.1. *Простая резистивная схема*

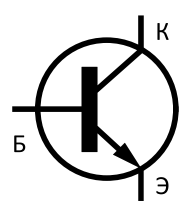


Рис. 2.2. *Схематическое обозначение транзистора типа n-p-n*

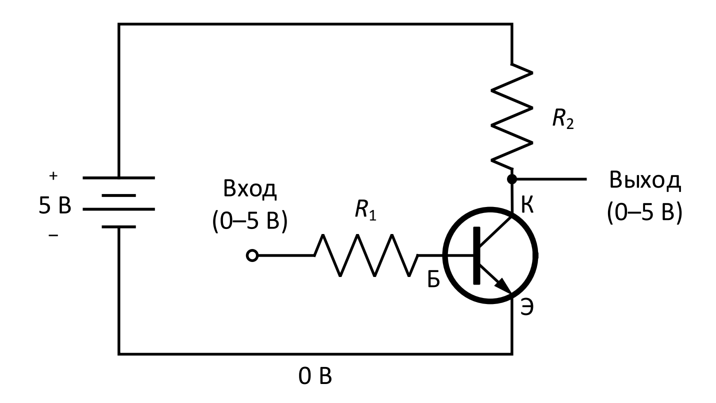


Рис. 2.3. *Транзисторный вентиль НЕ*

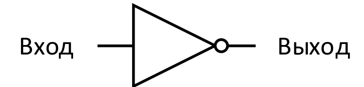


Рис. 2.4. *Схематическое обозначение вентиля НЕ*

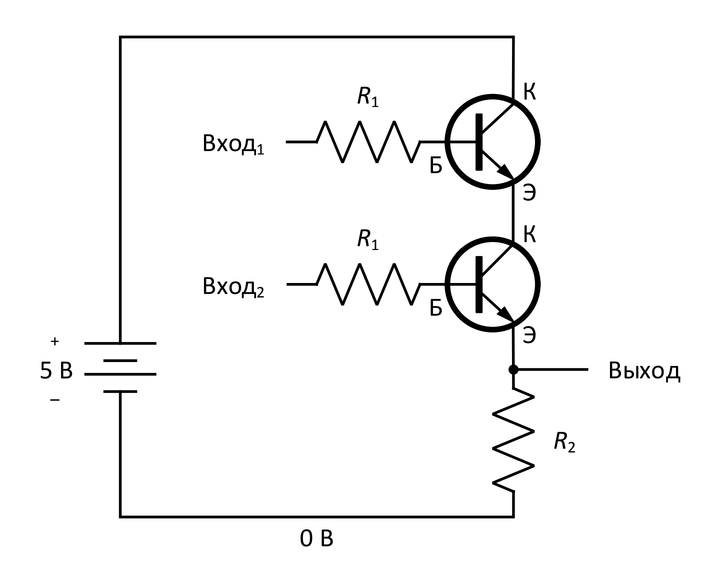


Рис. 2.5. *Транзисторный вентиль И*

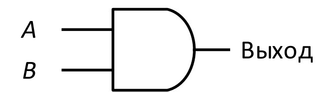


Рис. 2.6. *Схематическое обозначение вентиля И*

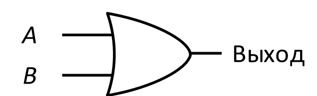


Рис. 2.7. *Схематическое обозначение вентиля ИЛИ*

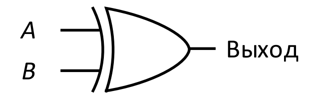


Рис. 2.8. *Схематическое обозначение вентиля исключающего ИЛИ*

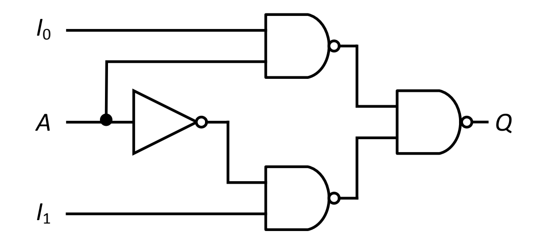


Рис. 2.9. *Схема мультиплексора с двумя входами*

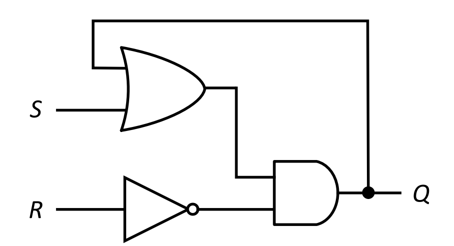


Рис. 2.10. *Схема SR-защелки*

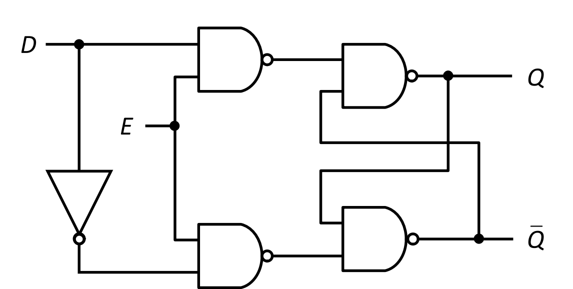


Рис. 2.11. *Схема управляемой D-защелки*

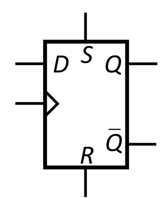


Рис. 2.12. *Схематическое обозначение D-триггера*

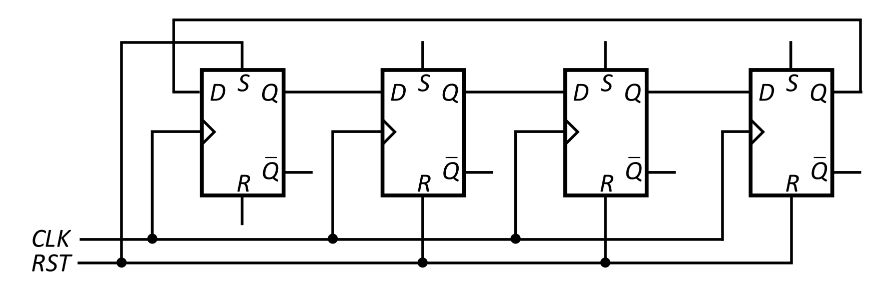


Рис. 2.13. *Схема четырехпозиционного кольцевого счетчика*

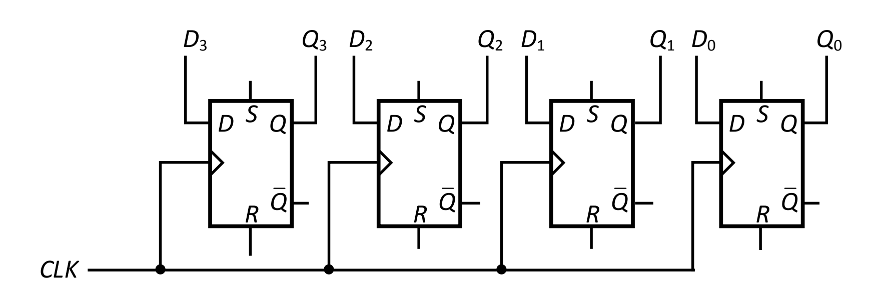


Рис. 2.14. *Схема 4-разрядного регистра*

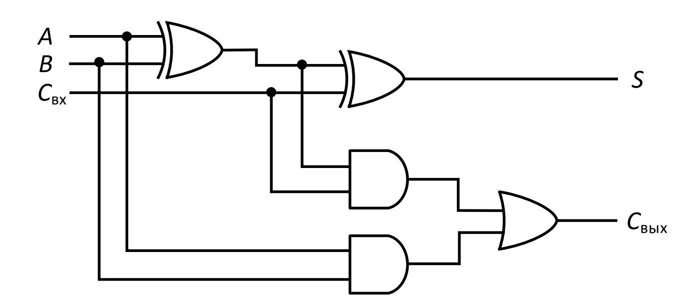


Рис. 2.15. *Схема полного сумматора*

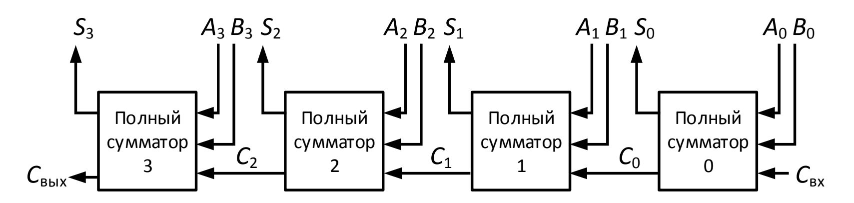


Рис. 2.16. *Схема 4-разрядного сумматора*

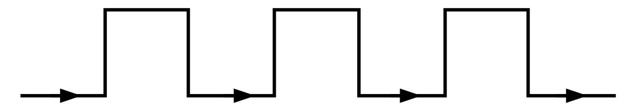


Рис. 2.17. *Прямоугольный сигнал*

3

Элементы процессора

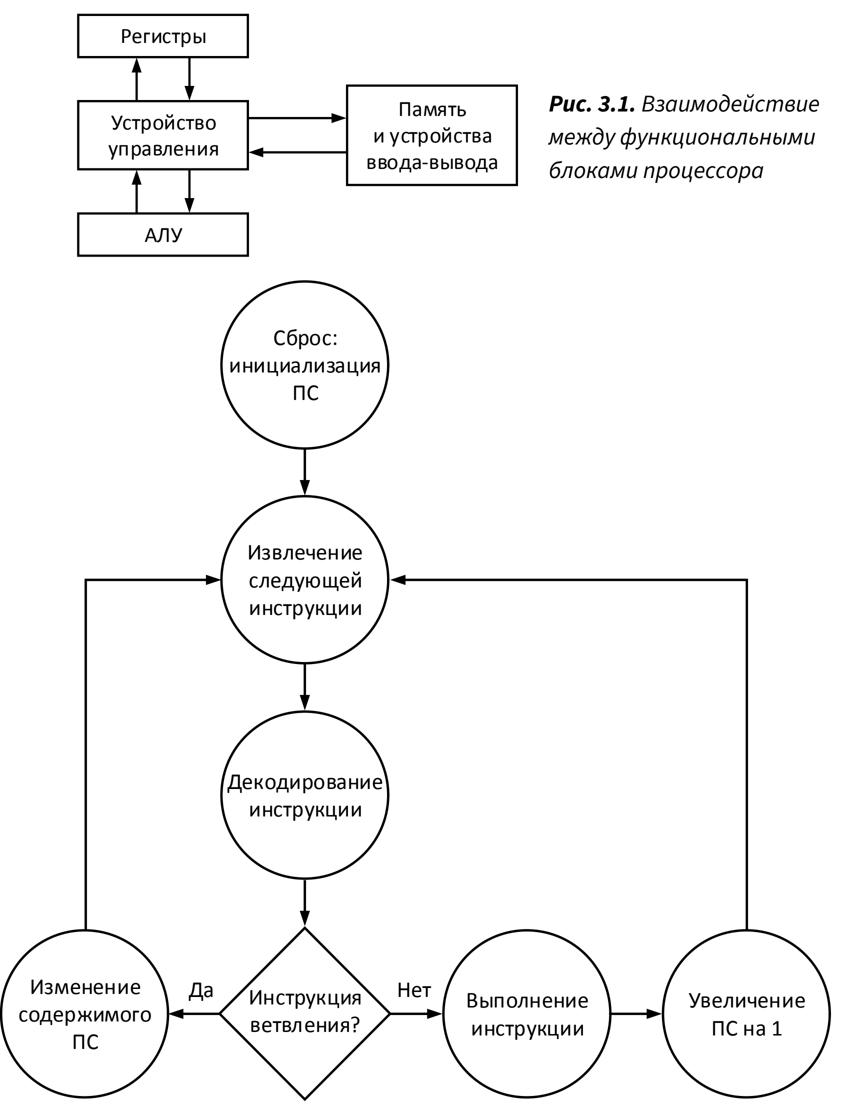


Рис. 3.1. *Взаимодействие между функциональными блоками процессора*

Рис. 3.2. *Цикл выполнения инструкции*

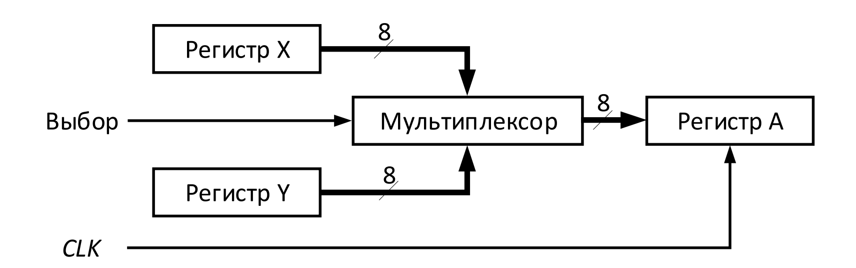


Рис. 3.3. *Инструкции TXA и TYA процессора 6502*

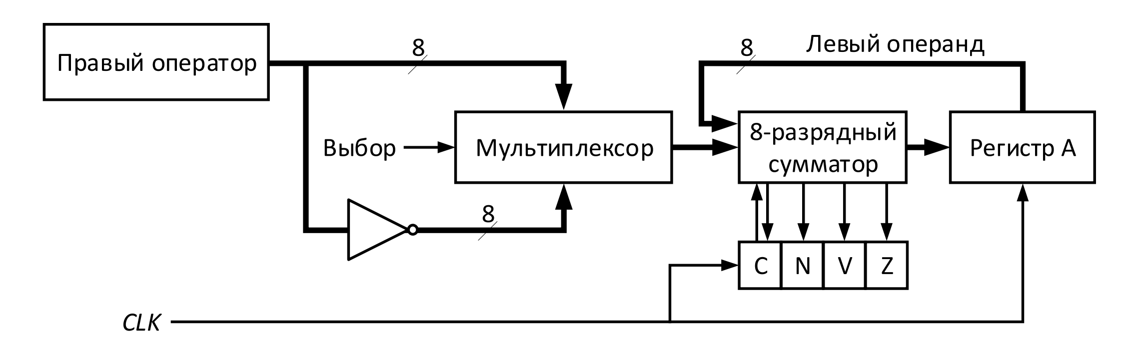


Рис. 3.4. *Операции сложения и вычитания в процессоре 6502*

4

Компоненты компьютерной системы

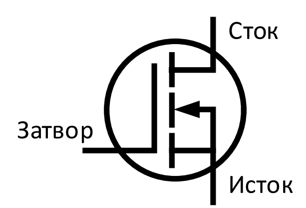


Рис. 4.1. *n-Канальный МОП-транзистор в режиме обогащения носителями*

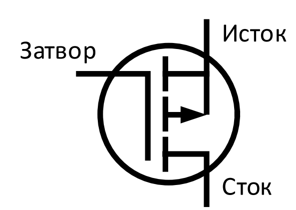


Рис. 4.2. *p-Канальный МОП-транзистор в режиме обогащения носителями*

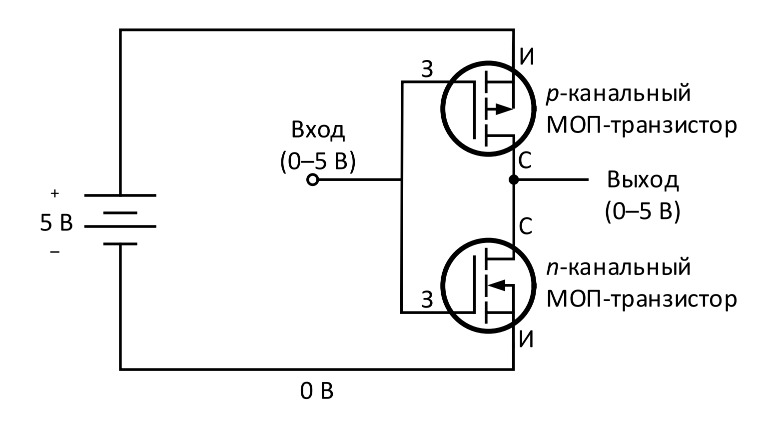


Рис. 4.3. *КМОП-схема вентиля НЕ*

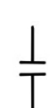


Рис. 4.4. *Схематическое обозначение конденсатора*

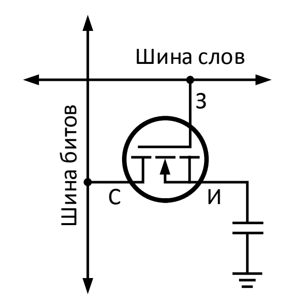


Рис. 4.5. *Схема битовой ячейки DRAM*

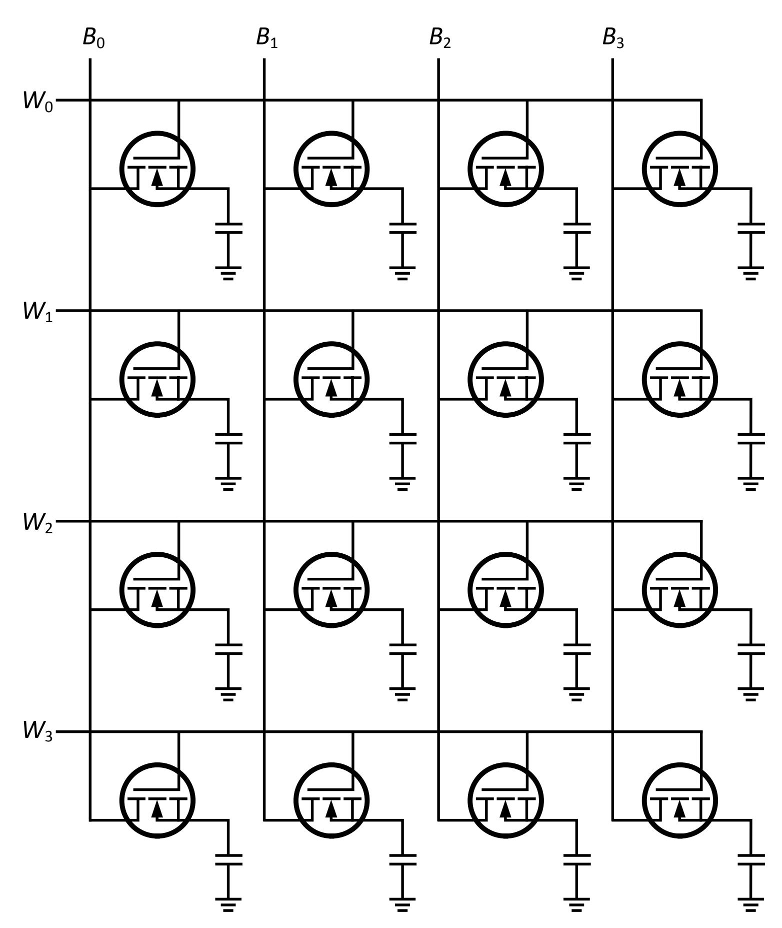


Рис. 4.6. *Организация банка памяти DRAM*

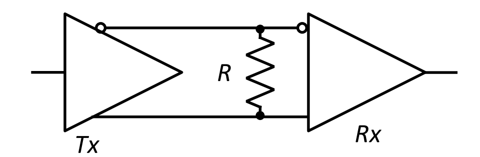


Рис. 4.7. *Схема последовательной шины с дифференциальной передачей сигналов*



5 Аппаратно-программный интерфейс

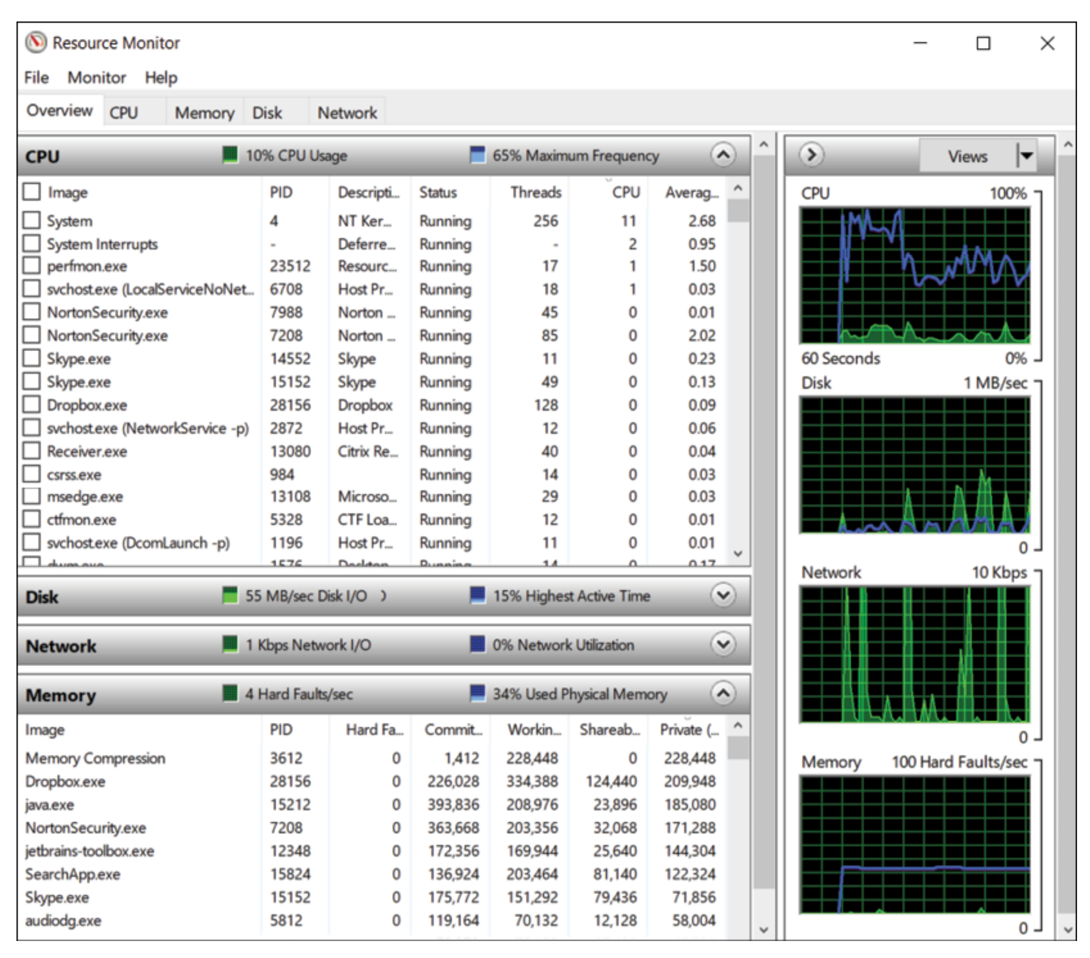


Рис. 5.1. *Экран сведений о процессах в Resource Monitor Windows*

```
jim@jim-VirtualBox: ~
top - 18:16:25 up 11:16,  1 user,  load average: 0.00, 0.21, 0.67
Tasks: 183 total,   1 running, 182 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.8 us,  1.0 sy,  0.0 ni, 94.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4046384 total,  1135716 free,   900736 used,  2009932 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  2798540 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 2110 jim       20   0 1224192 163008  81456 S  3.7  4.0 508:13.96 compiz
  974 root      20   0  358516  79944  36340 S  2.3  2.0  88:58.41 Xorg
 2259 jim       20   0  945448  49416  39064 S  0.7  1.2   0:11.74 nautilus
 9456 jim       20   0  665284  36824  29000 S  0.7  0.9   0:00.84 gnome-terminal-
 1808 jim       20   0  111968   2132   1780 S  0.3  0.1   0:34.57 VBoxClient
 1882 jim       20   0  344984   6452   5376 S  0.3  0.2   0:00.07 ibus-daemon
    1 root      20   0  119876   5992   3980 S  0.0  0.1   0:01.50 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:00.26 ksoftirqd/0
    5 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S  0.0  0.0   0:01.30 kworker/u2:0
    7 root      20   0       0      0      0 S  0.0  0.0   0:01.31 rcu_sched
    8 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S  0.0  0.0   0:00.00 migration/0
   10 root      rt   0       0      0      0 S  0.0  0.0   0:00.10 watchdog/0
   11 root      20   0       0      0      0 S  0.0  0.0   0:00.00 kdevtmpfs
   12 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 netns
   13 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 perf
   14 root      20   0       0      0      0 S  0.0  0.0   0:00.01 khungtaskd
```

Рис. 5.2. *Экран с информацией о процессах Linux, выведенной по команде* `top`

6

Специализированные вычисления

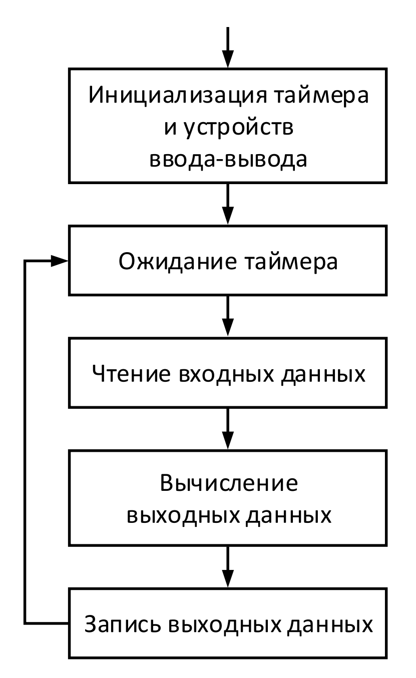


Рис. 6.1. *Поток управления системой реального времени*

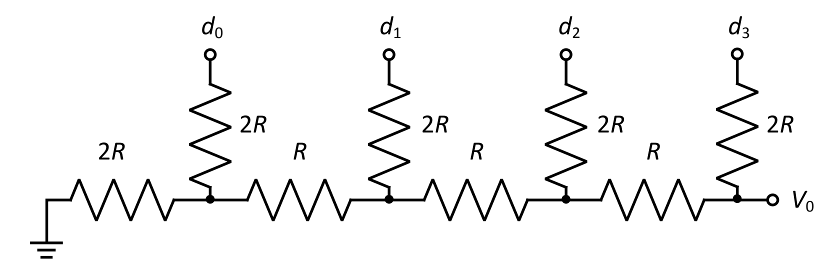


Рис. 6.2. *ЦАП лестничного типа R–2R*

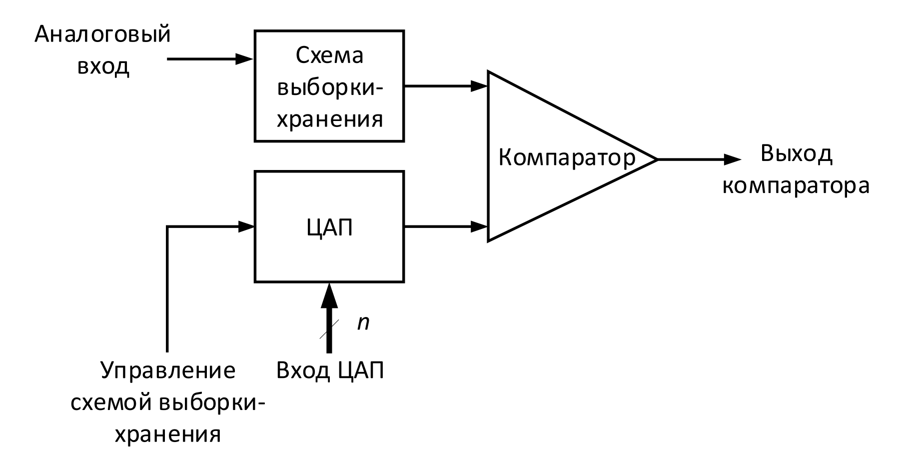


Рис. 6.3. *Архитектура АЦП*

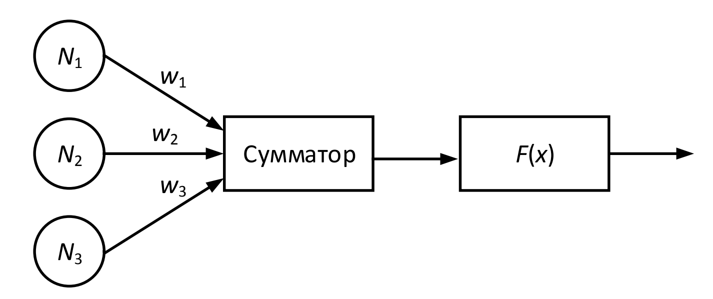


Рис. 6.4. *Нейрон, получающий входные сигналы от трех других нейронов*

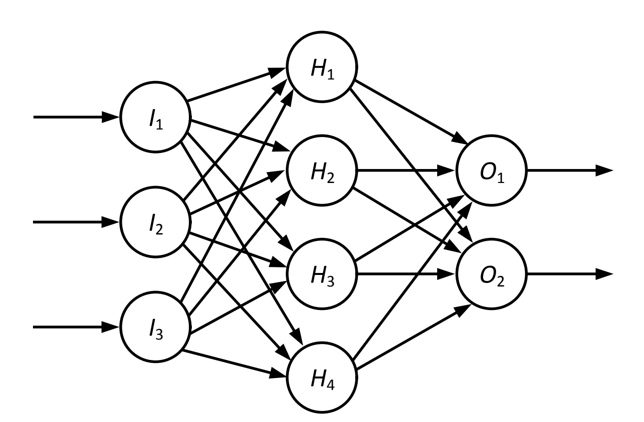


Рис. 6.5. *Трехуровневая сеть прямого распространения*

7

Архитектура процессора и памяти

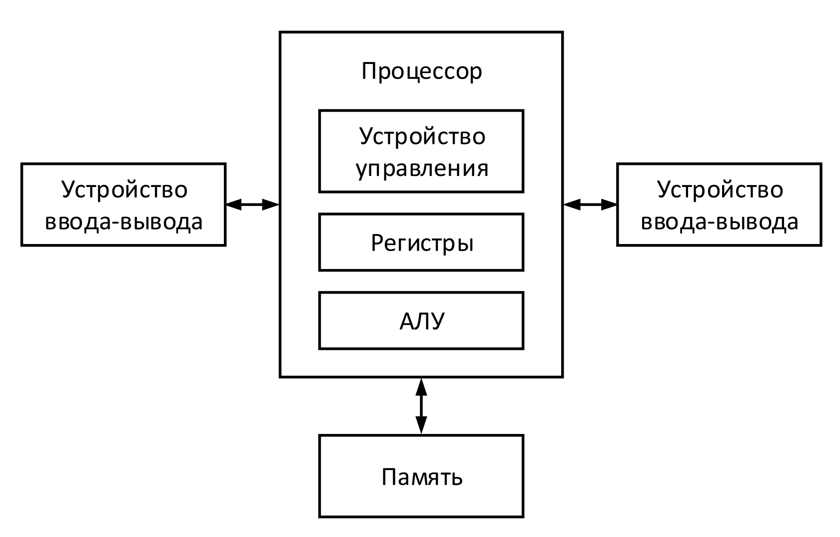


Рис. 7.1. *Фон-неймановская архитектура*

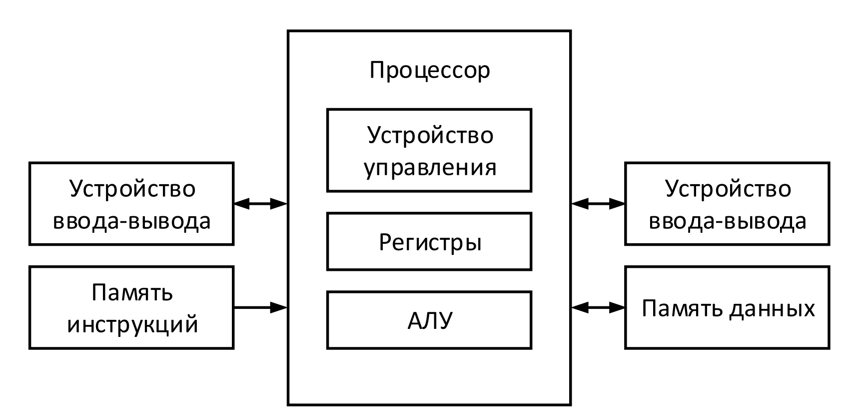


Рис. 7.2. *Гарвардская архитектура*

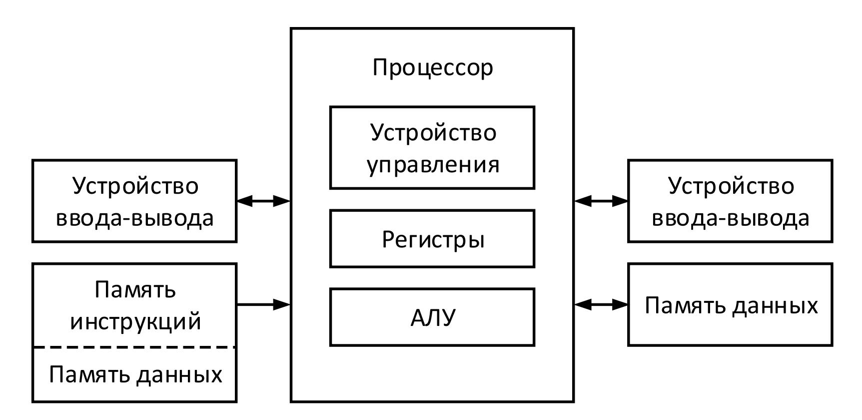


Рис. 7.3. *Модифицированная гарвардская архитектура*

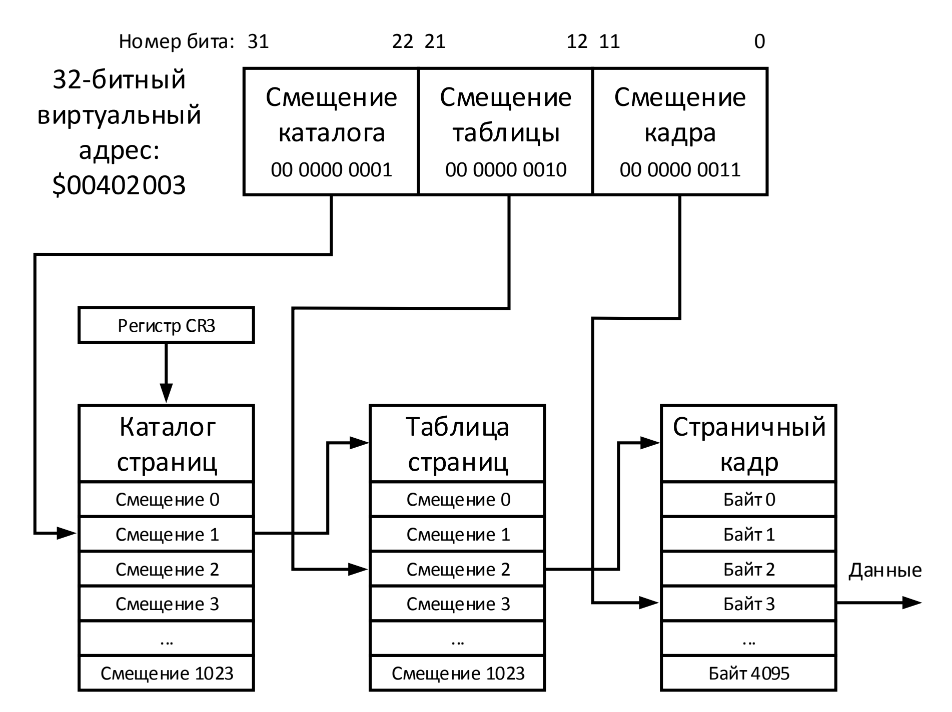


Рис. 7.4. *Трансляция виртуального адреса в физический*

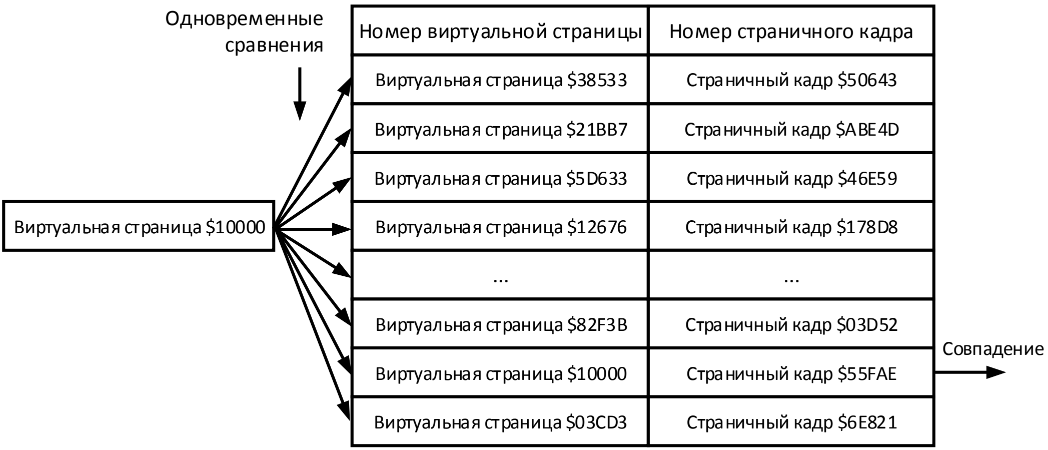
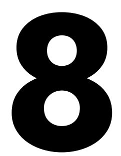


Рис. 7.5. *Работа буфера ассоциативной трансляции*



Методы повышения производительности

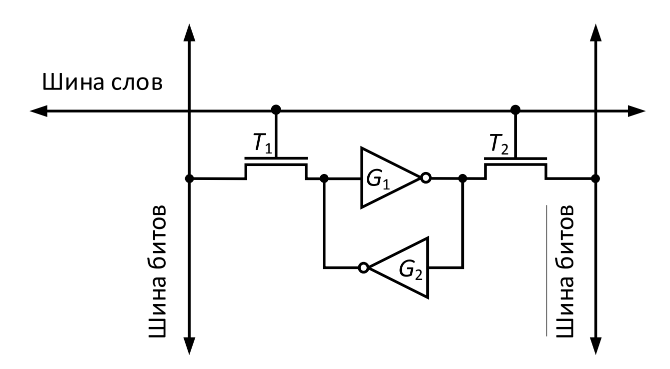


Рис. 8.1. *Схема ячейки SRAM*

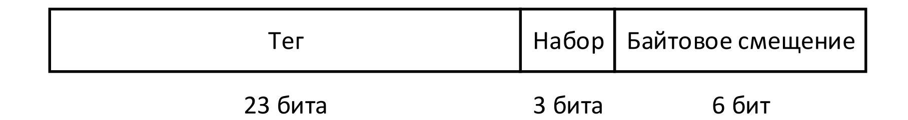


Рис. 8.2. *Компоненты 32-разрядного адреса физической памяти*

32-разрядный физический адрес

Тег | Набор | Байтовое смещение

23 бита | 3 бита | 6 бит

Набор 000b
Набор 001b
Набор 010b
Набор 011b
Набор 100b
Набор 101b
Набор 110b
Набор 111b

Допуст. | Тег | Данные

1 бит | 23 бита | 64 байта

Рис. 8.3. *Связь 32-разрядного физического адреса с кэш-памятью*

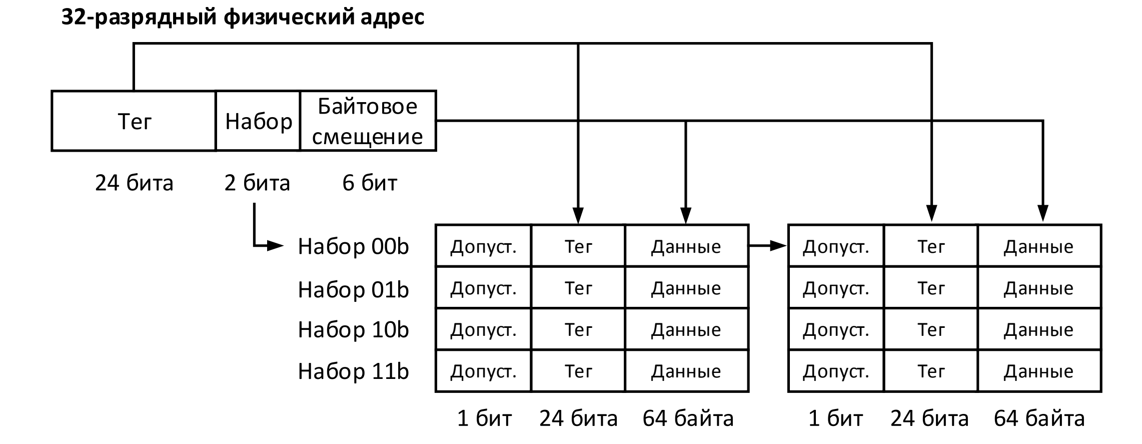


Рис. 8.4. *Работа наборно-ассоциативного кэша*

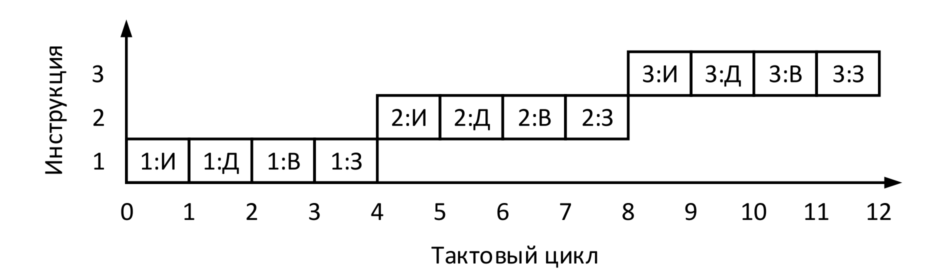


Рис. 8.5. *Последовательное выполнение инструкций*

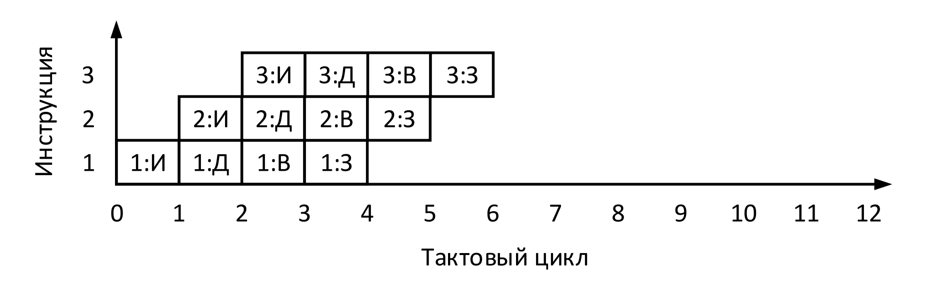


Рис. 8.6. *Конвейерное выполнение инструкций*

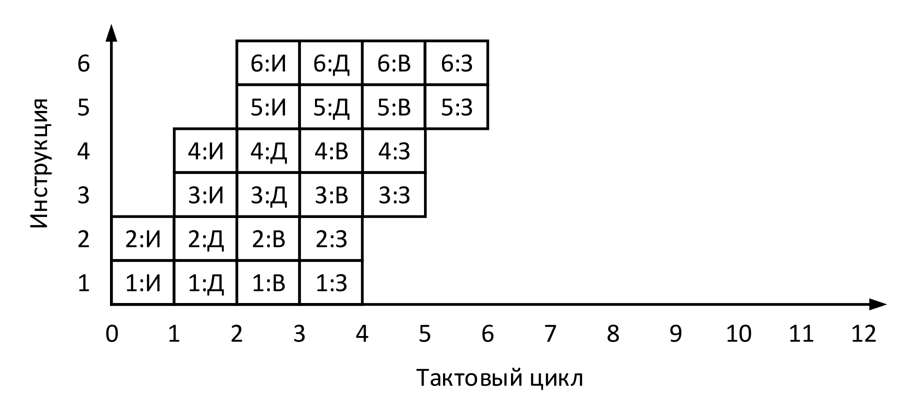


Рис. 8.7. *Конвейерное выполнение с множественной выдачей инструкций*



Специализированные расширения процессоров

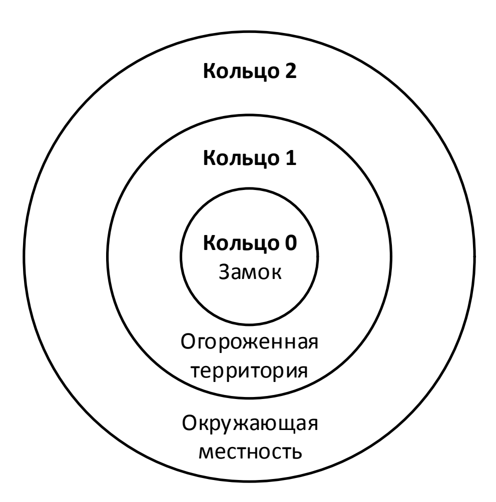


Рис. 9.1. *Пример защитных колец*



Рис. 9.2. *Кольца защиты в операционных системах x86*

	Знак	Порядок	Мантисса
Короткое действительное число (32 бита)	1 бит	8 бит	23 бита
Длинное действительное число (64 бита)	1 бит	11 бит	52 бита
Временное действительное число (80 битов)	1 бит	15 бит	64 бита

Рис. 9.3. *Форматы данных с плавающей запятой в сопроцессоре 8087*

10

Современные архитектуры и наборы инструкций процессоров

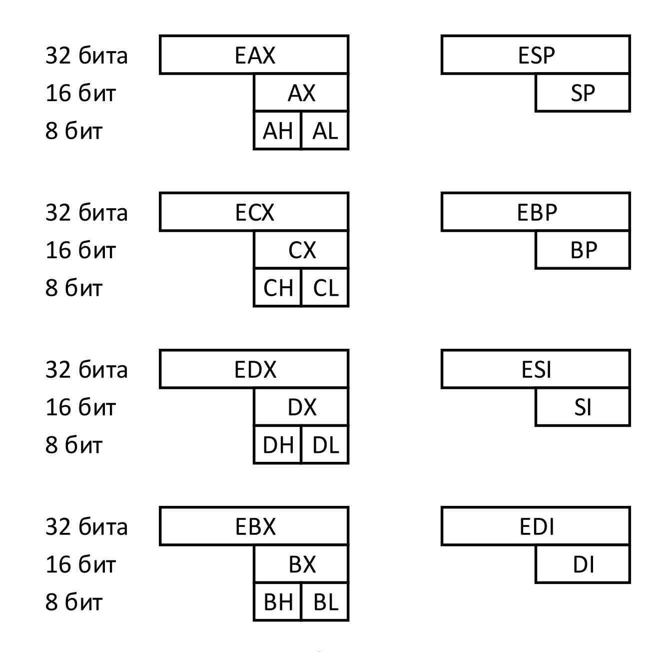


Рис. 10.1. *Имена и подмножества регистров*

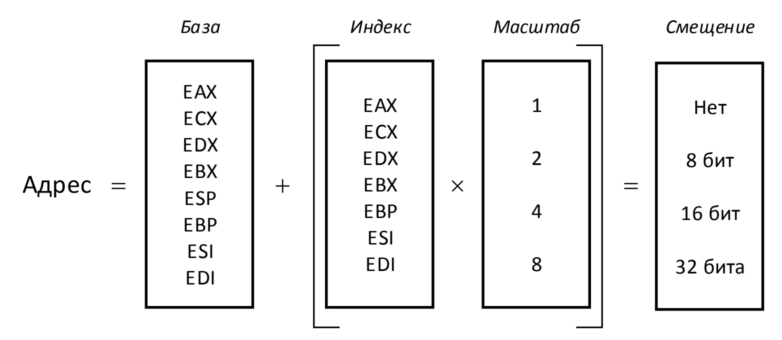


Рис. 10.2. *Режим относительной адресации*

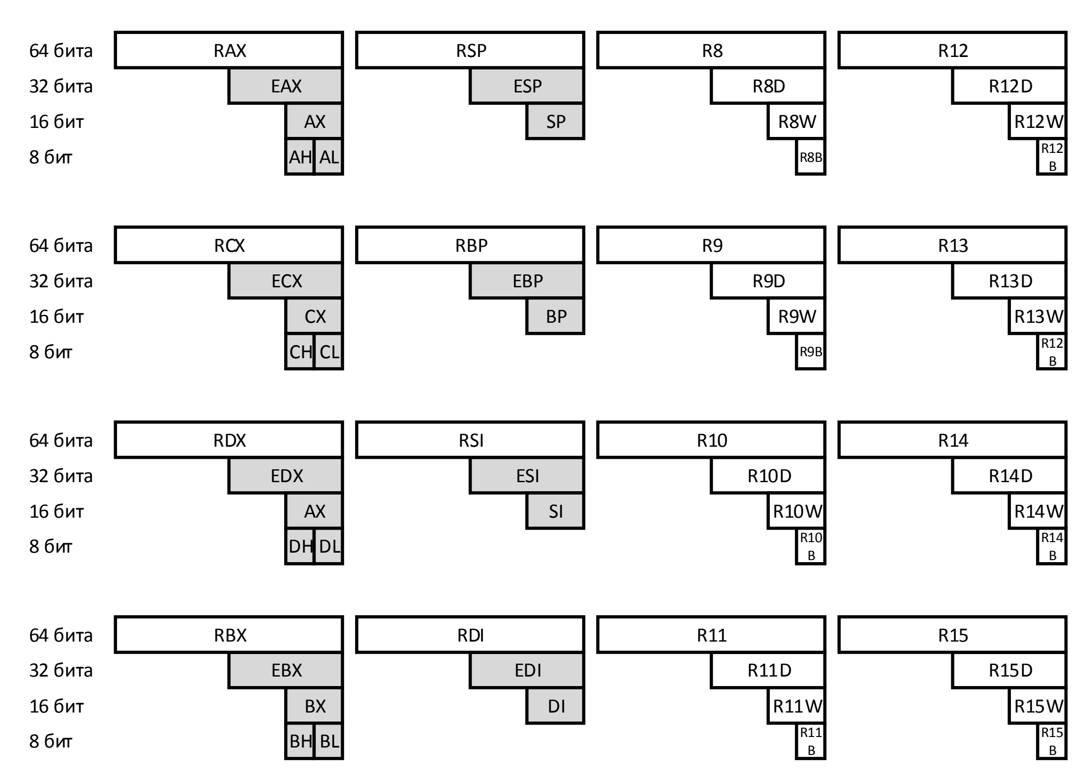


Рис. 10.3. *Регистры архитектуры x64*

11

Архитектура и набор инструкций RISC-V

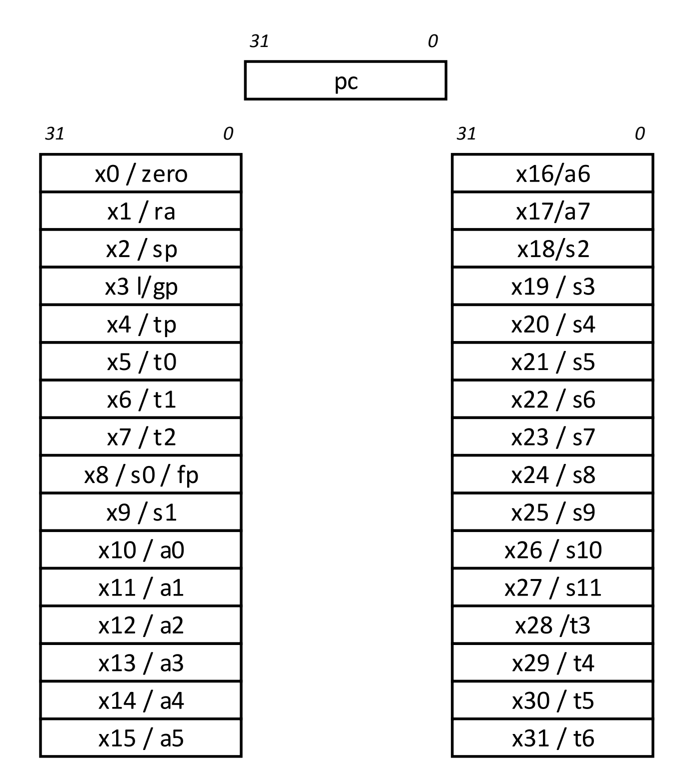


Рис. 11.1. *Базовый набор регистров ISA RISC-V*

13

Специализированные компьютерные архитектуры

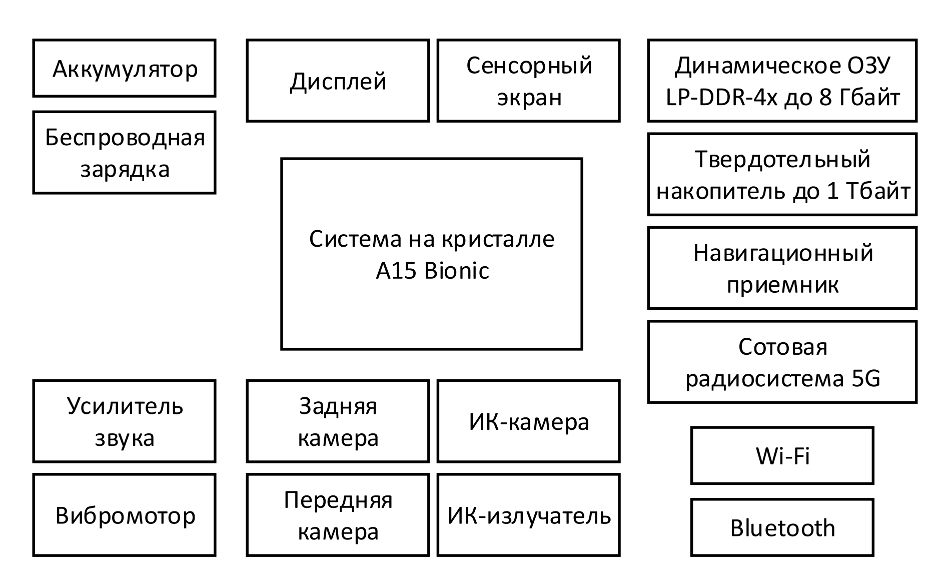


Рис. 13.1. *Компоненты iPhone 13 Pro Max*

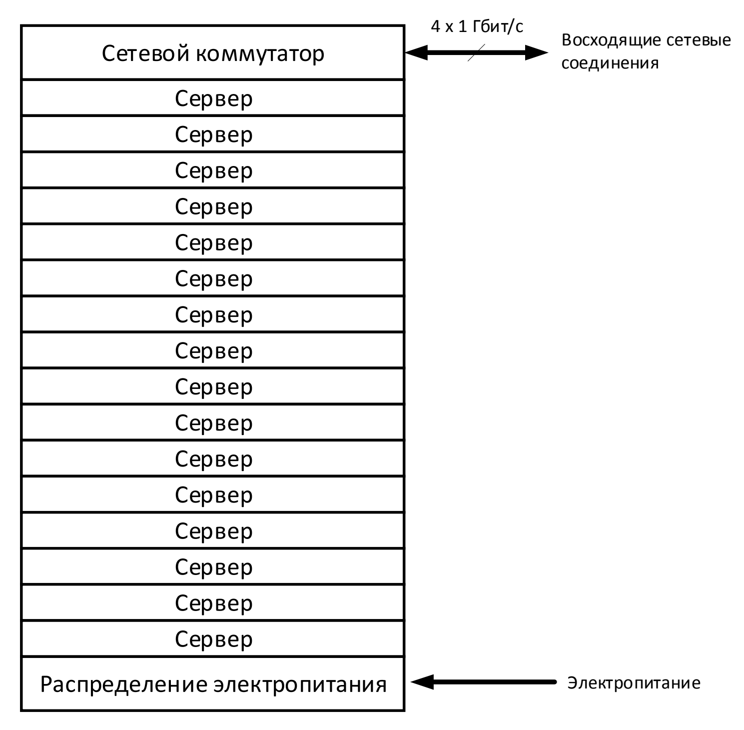


Рис. 13.2. *Стойка с 16 серверами*

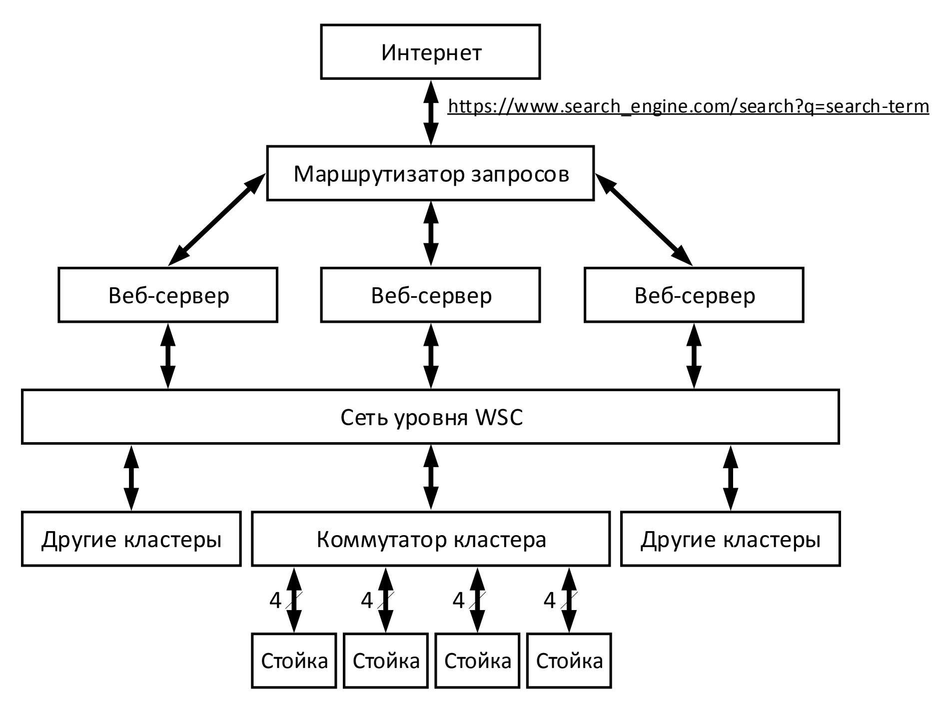


Рис. 13.3. *Внутренняя сеть WSC*

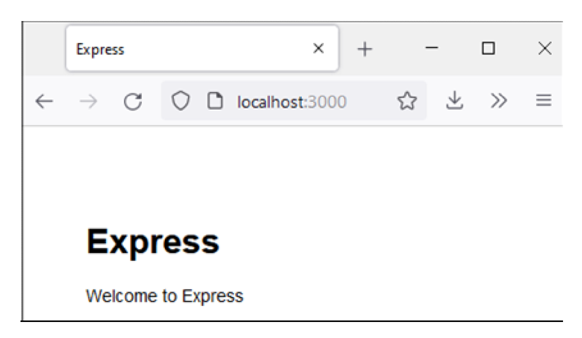


Рис. 13.4. *Экран простого приложения Node.js*

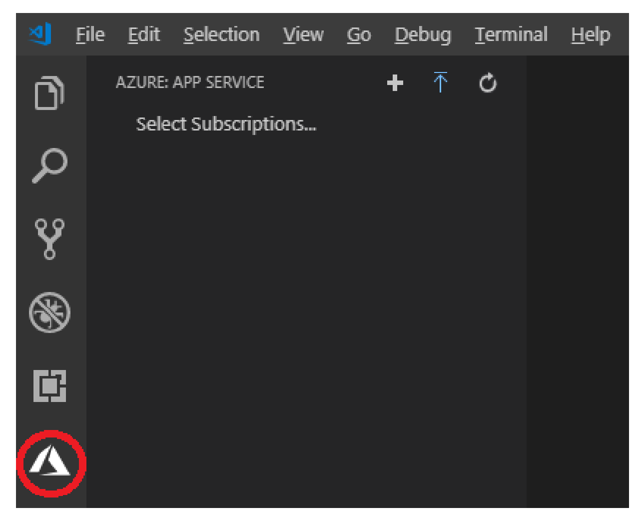


Рис. 13.5. *Добавление параметра приложения*

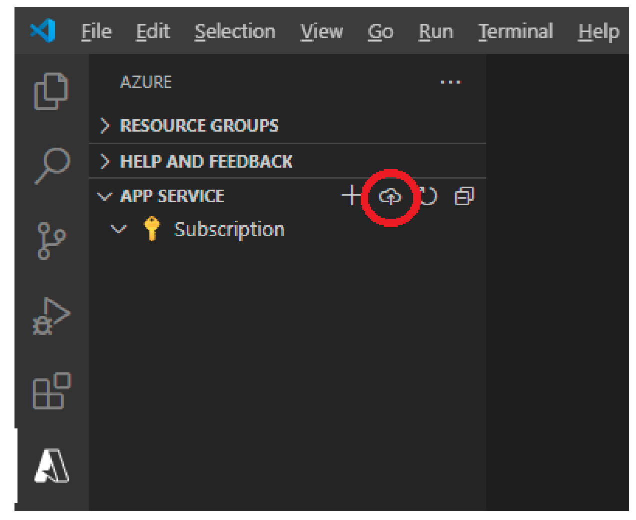


Рис. 13.6. *Развертывание в облаке*

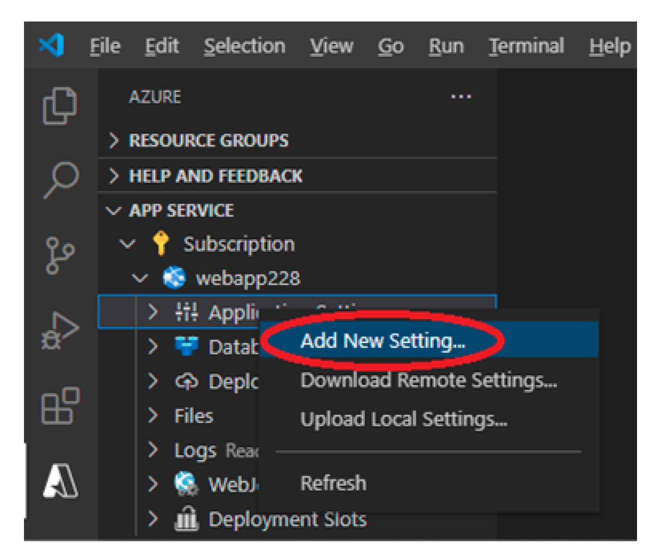


Рис. 13.7. *Добавление параметра приложения*

15

Архитектуры блокчейна и майнинга биткоинов

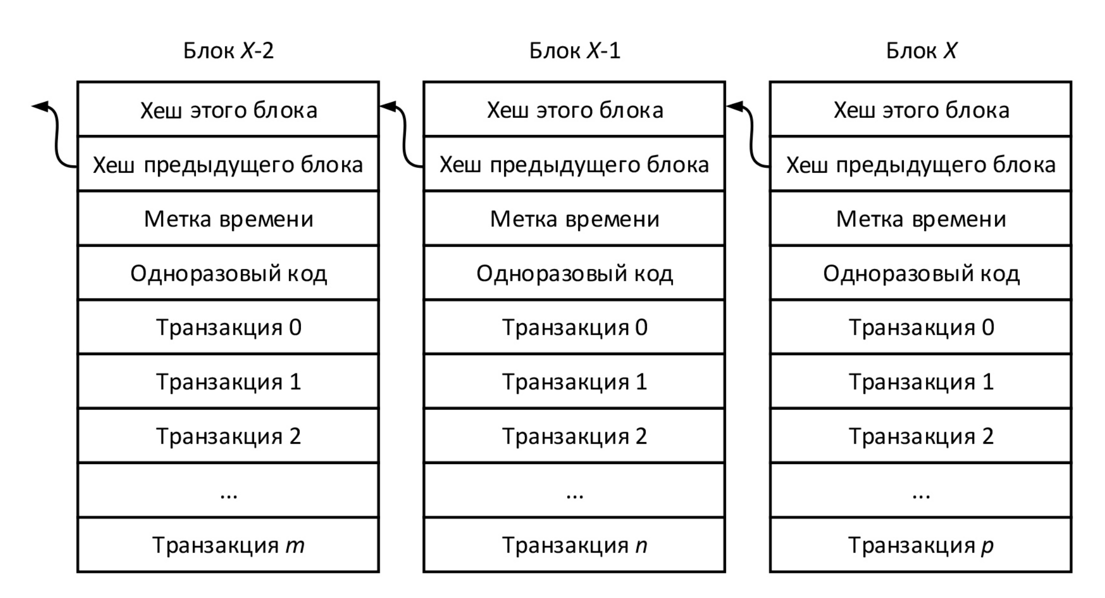


Рис. 15.1. *Упрощенное представление блокчейна*

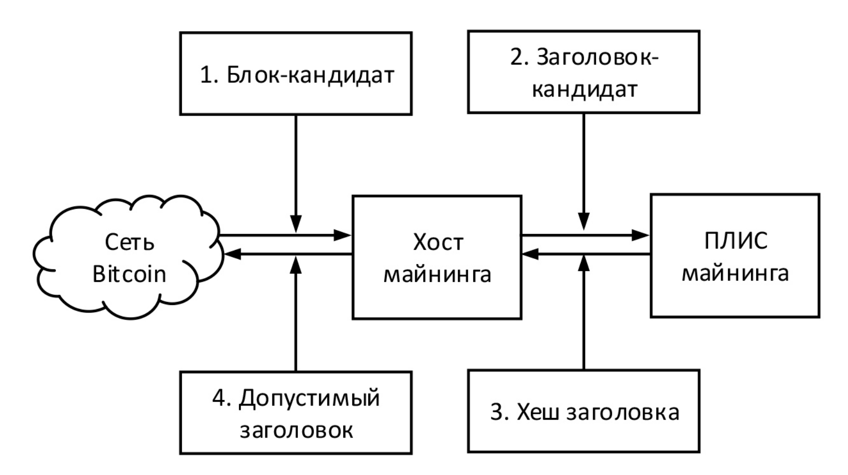


Рис. 15.2. *Простая реализация майнинга на основе ПЛИС*

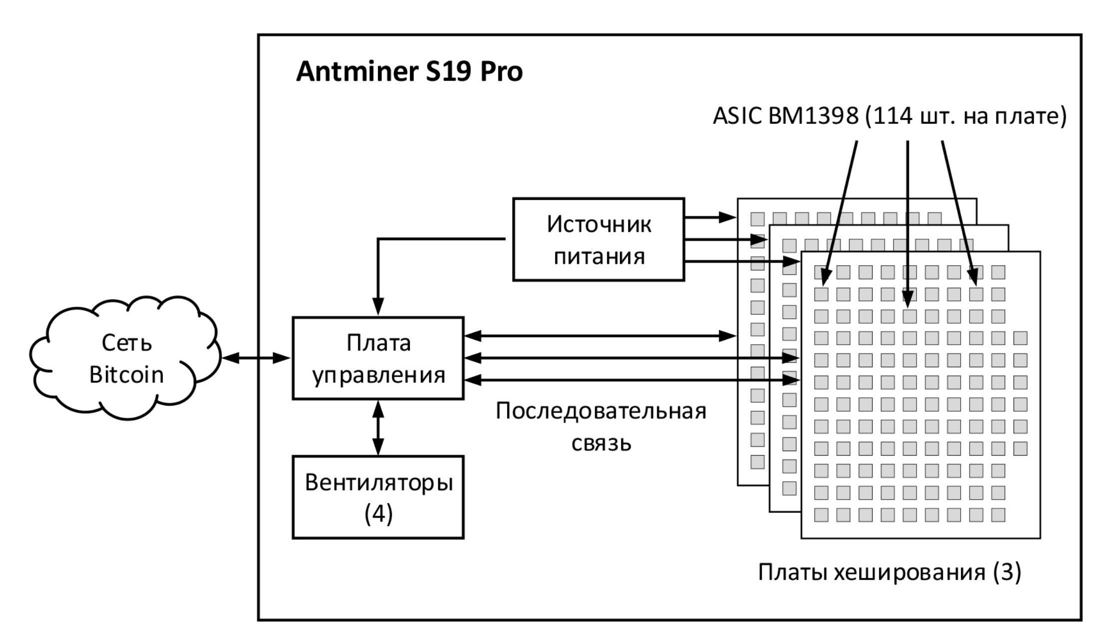


Рис. 15.3. *Конфигурация аппаратных средств Antminer S19 Pro*

16

Архитектуры для самоуправляемых автомобилей



Рис. 16.1. *Уровни CNN, используемые для идентификации объектов*

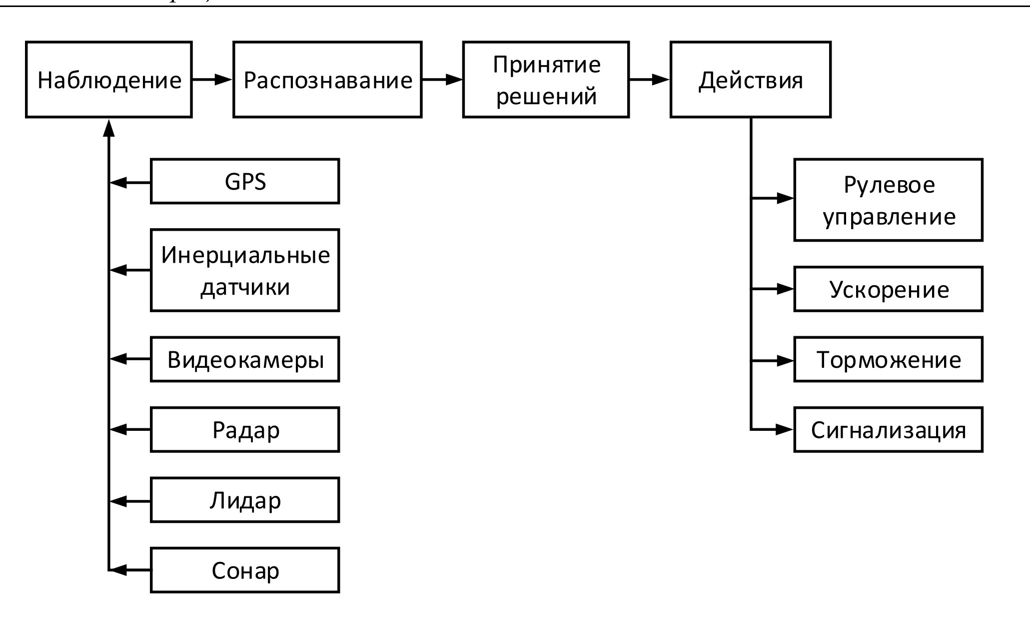


Рис. 16.2. *Компоненты и процессы автономной системы вождения*

Приложение
Ответы к упражнениям

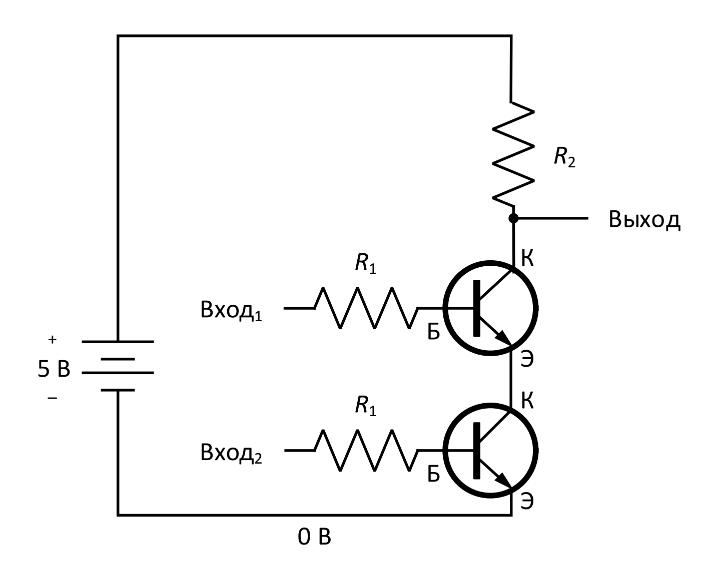


Рис. П1. *Схема вентиля И-НЕ*

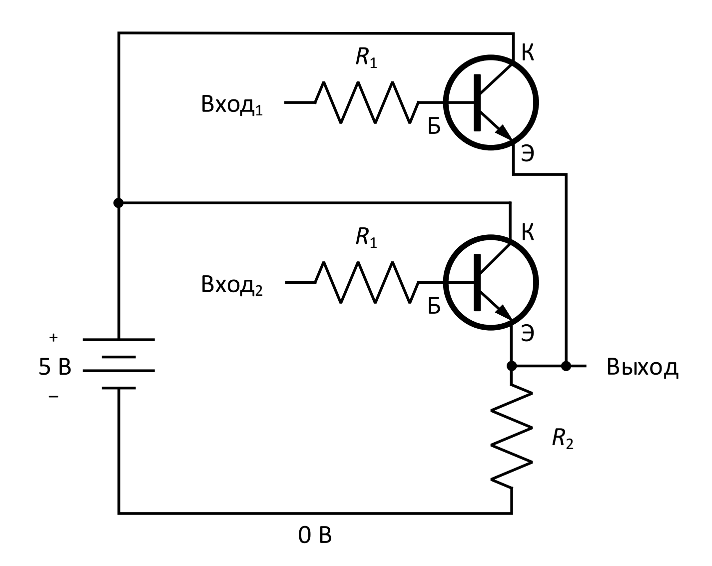


Рис. П2. *Схема вентиля ИЛИ*

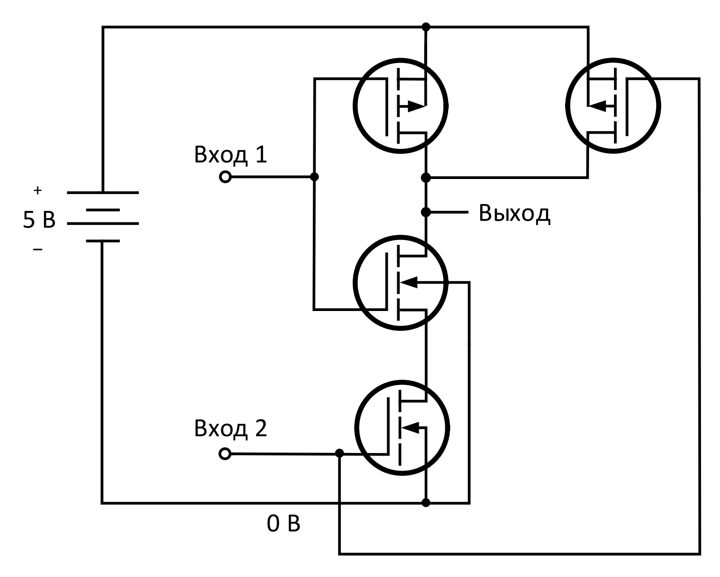


Рис. П3. *Схема вентиля И-НЕ*

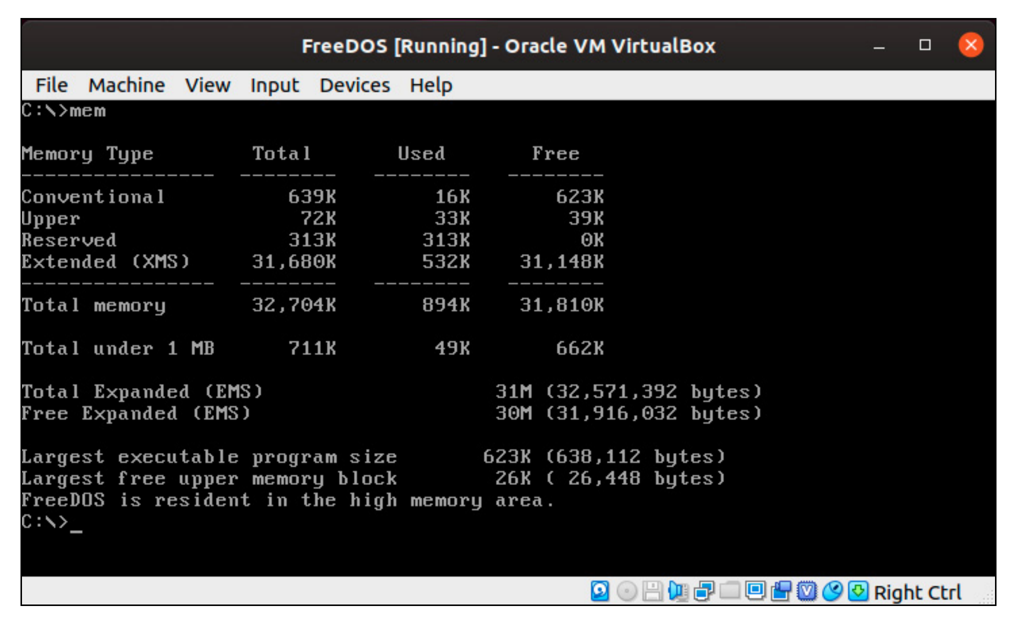


Рис. П4. *Копия экрана FreeDOS*

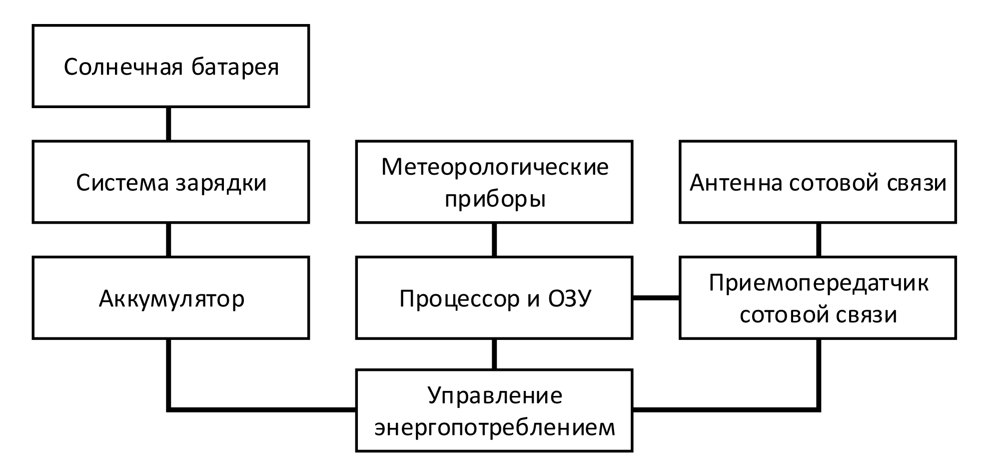


Рис. П5. *Начальная схема системы сбора данных о погоде*

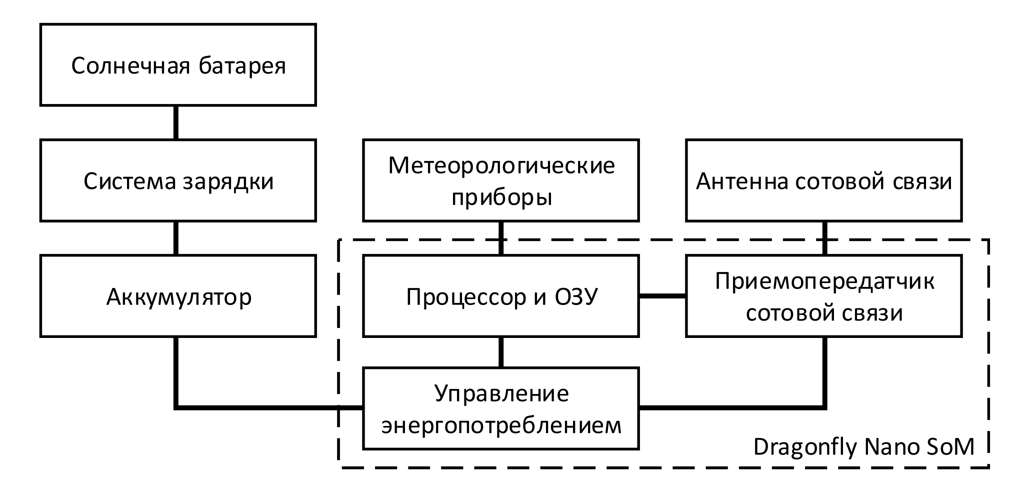


Рис. П6. *Окончательная схема системы сбора данных о погоде*

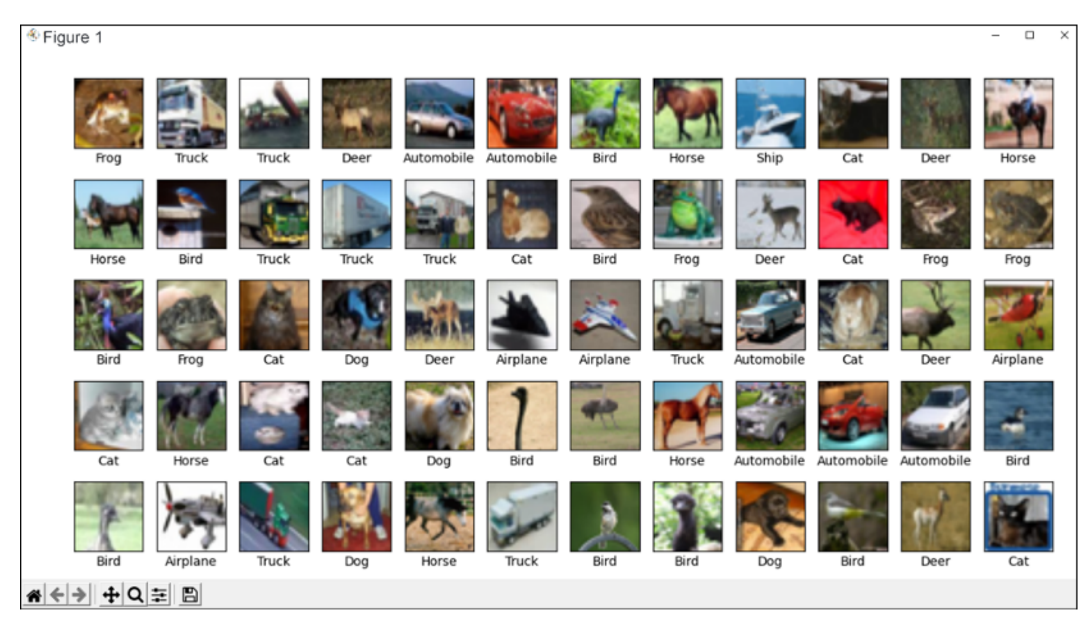


Рис. П7. *Примеры изображений набора данных CIFAR*



Рис. П8. *Структура CNN для классификации изображений*

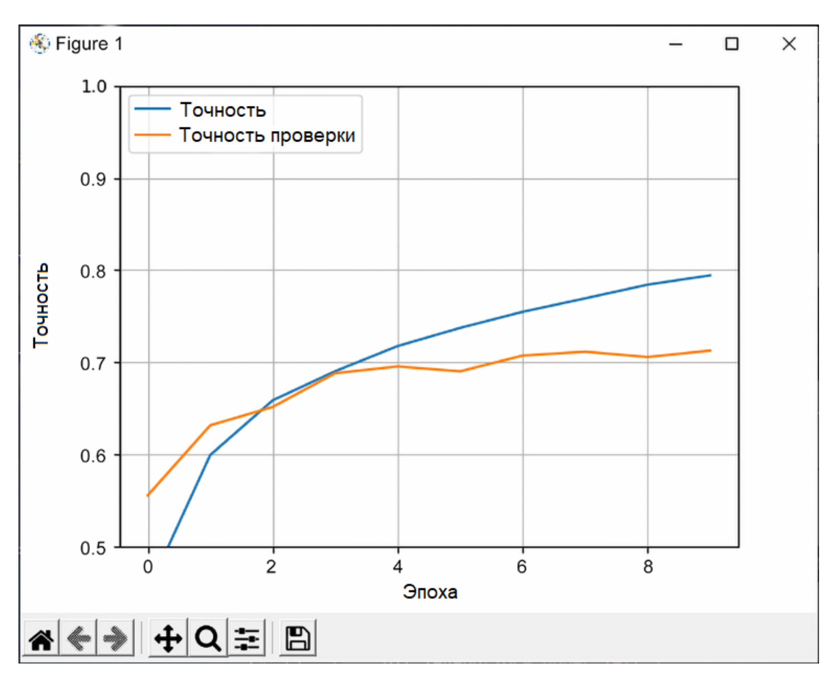


Рис. П9. *Точность классификации изображений сетью CNN*